1986

Silicon compilation using a LISP-based layout language.

Malagon-Fajar, Manuel Ambrosio

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

SILICON COMPILATION USING A LISP-BASED LAYOUT LANGUAGE

by

Manuel Ambrosio Malagon-Fajar

June 1986

Thesis Advisor: D. E. Kirk

CURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION NCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION aval Postgraduate School	6b OFFICE SYMBOL (If applicable) 62
7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School	
6c ADDRESS (City, State, and ZIP Code) onterey, California 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
8a NAME OF FUNDING / SPONSORING ORGANIZATION	8b OFFICE SYMBOL (If applicable)
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
ILICON COMPILATION USING A LISP-BASED LAYOUT LANGUAGE

12 PERSONAL AUTHOR(S)
anuel Ambrosio Malagon-Fajar

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
aster's Thesis	FROM _____ TO _____	86 June	208

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Silicon Compilation; LISP; Layout Language

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

Two related silicon compilers developed at IT's Lincoln Laboratory with a common layout language are examined. he simpler one, the Lincoln Boolean Synthesizer (LBS), is a Complementary etal Oxide (CMOS) technology based program for generating chips out of rbitrary boolean expressions. MacPitts, on the other hand, can nplement advanced programming language constructs in N-Channel (NMOS) echnology. A study of their layout language, Lincoln Laboratory's ISP-based Layout Language (L5), and its implementation is presented. n addition, there is also a brief discussion of how Macpitts's functional epertoire can be changed.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED

22a NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE (Include Area Code)	22c OFFICE SYMBOL
Prof D. E. Kirk	(408)646-3451	62Ki

FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

Approved for public release; distribution is unlimited.

Silicon Compilation Using a LISP-based Layout Language

by

Manuel Ambrosio Malagón-Fajar
Lieutenant Commander, United States Navy
B.S., Massachusetts Institute of Technology, 1974

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

ABSTRACT

Two related silicon compilers developed at MIT's Lincoln Laboratory with a common layout language are examined. The simpler one, the Lincoln Boolean Synthesizer (LBS), is a Complementary Metal Oxide (CMOS) technology based program for generating chips out of arbitrary boolean expressions. MacPitts, on the other hand, implements advanced programming language constructs in N-Channel (NMOS) technology. A study of their layout language, Lincoln Laboratory's LISP-based Layout Language (L5), and its use is presented. In addition, there is also a brief discussion of how Macpitts' functional repertoire can be changed.

TABLE OF CONTENTS

I. INTRODUCTION 12

A. BACKGROUND 12

B. SCOPE OF THIS THESIS INVESTIGATION 14

II. THE LISP ENVIRONMENT 18

A. LISP: A FUNCTIONAL PROGRAMMING LANGUAGE 18

1. Functional Programming 19

2. Backus Naur Format (BNF) 19

3. Lambda Functions 21

4. Variable Scoping 24

5. Recursion and Iteration 27

B. INTERPRETED, COMPILED OR DUMPED LISP 29

1. The LISP Read-Eval-Print Loop: Interpreted LISP 29

a. The LISP Prompt " -> ", Start " (" and Stop ") " 29

b. A Universal LISP Function: **eval** 30

c. Eval's Dual: **quote** or " ' " 31

2. Compiled LISP 33

a. The Compiler 33

b. Compilation Dependencies: Makefile 34

3. Interpreted and Compiled LISP: **dumplisp** 37

4. Basic Input and Output (I/O) 40

C. LISP FUNCTIONS AND DATA 42

1. LISP's Basic Structure: The List 43

2. LISP Function Definition: **def** and **defun** 43
3. Frequently Used LISP Functions 47
a. Binding Variables: **set**, **setq**, **let** and **let*** 47
b. List Selection: **car**, **cdr**, **nth** and **nthcdr** 49
c. List Construction: **cons**, **append** and **list** 51
d. Functional Application: **apply** and **funcall** 52
e. Predicates(the Values **t** and **nil**) and the **cond** Control Structure 53
f. Iteration: **prog**, **do**, **do*** and **mapcar** 53
4. Iteration and Recursion 60
D. THE FRANZ LISP PROGRAMMING ENVIRONMENT 62
1. Program Development Aids 62
2. Summary 68
III. MACROS, FUNCTIONS AND DATA STRUCTURES: LINCOLN.L 70
A. MACROS 70
1. Data Abstraction and Macros 71
2. Eval and the Backquote Macro 72
3. Lincoln.l Macros 80
a. Numerical Comparison Predicate Macros 80
b. Type Predicate Macros 81
B. FUNCTIONS 83
1. APL Like Operators 83
2. Selection Functions 84
3. Set Functions 85
4. Numeric Functions 86

C. DEFSTRUCTS 87
1. Short **defstructs** 87
a. Short Constructor 88
b. Short Selector 89
c. Short Mutator 89
2. Long **defstructs** 90
a. Long Constructor 91
b. Long Selector 92
c. Long Mutator 92
d. Long Interrogator 93
3. General Field Structure Checks 94
4. Summary 95
IV. LINCOLN LABORATORY LISP LAYOUT LANGUAGE: L5 96
A. GLOBAL VARIABLES AND DATA TYPES 96
1. Global Variables 96
2. L5 Data Structures 101
B. ITEMS AND THEIR OPERATIONS 105
1. Item Creation 105
2. Item Operators 106
a. Translation and Merging Operators 106
b. Query Operators 112
c. Point Operators 114
d. River Router 122
C. HIERARCHICAL REPRESENTATION 124
1. Defsymbols 125

a. **in-memory** storage 126
b. **on-disk** storage 129
2. CIF 131
3. Caesar 132
4. Summary 134
V. TOP.L AND PREPASS.L: THE TOP-LEVEL 135
A. THE TOP-LEVEL 135
1. Franz Lisp's Default Top-Level 135
2. Example Top-Level 136
B. LBS COMPILER 142
C. MACPITTS COMPILER 145
VI. ORGANELLES 153
A. OVERVIEW 153
B. AN EXAMPLE 159
VII. CONCLUSIONS 169
APPENDIX A : MISCELLANEOUS TOPICS 171
A. LAYOUT ERRORS 171
B. EXPERIMENTING IN MACPITTS 174
LIST OF REFERENCES 182
BIBLIOGRAPHY 185
INITIAL DISTRIBUTION LIST 204

LIST OF TABLES

2.1 EXAMPLE LISP OBJECTS 31

3.1 BACKQUOTE MACRO SYMBOLS 73

3.2 NUMERICAL COMPARISON PREDICATES 81

3.3 TYPE PREDICATES AND THEIR MEANINGS 82

3.4 DEFSTRUCT FUNCTION SUMMARY 95

4.1 GLOBAL VARIABLES AND THEIR FUNCTIONS 97

4.2 GLOBAL VARIABLE DEFAULT VALUES 98

4.3 AN ITEM'S SYNTAX 101

4.4 FOUR PRIMITIVE ITEM CREATING FUNCTIONS 105

4.5 TRANSLATION AND MERGING OPERATORS 107

4.6 ITEM QUERY OPERATORS 112

4.7 POINT OPERATORS 114

4.8 CAESAR FUNCTIONS 133

5.1 LBS SYNTAX 142

5.2 MACPITTS SYNTAX 145

5.3 PREPASS.L FUNCTION SYNTAX 148

6.1 MACPITTS PROGRAM FUNCTION SUMMARY 155

A.1 ORGANELLE SPECIFICATION COMPARISON 171

LIST OF FIGURES

2.1 Passing of Values in Nested Functions 24

2.2 The LISP Object Hierarchy 31

3.1 The Defstruct Function Hierarchy 75

3.2 " **man** " Defstruct Operator Functions 78

3.3 The **defstruct-long** Definition Without Backquote 79

4.1 **(layout-inverter 4 t)** 109

4.2 **(align-items inverter 'vdd (mirrorx inverter) 'vdd)** 111

4.3 **(merge inverter (move inverter 20 0))** 113

4.4 **(layout-and 4 4 t)** 116

4.5 **(layout-flag '(ini menie mini moe) 9 138)** 121

4.6 **(river 'NM 3 10 '(1 8 17 26 37) '(5 17 20 41 57))** 123

4.7 **LS-symboi-storage** 126

4.8 Caesar, LS and CIF Conversions 133

6.1 Prepass Function Flow 154

6.2 MacPitts Program Hierarchy 154

6.3 **definition defstruct** 158

6.4 **(organelle-==-bit-0)** 160

6.5 **(organelle-==-bit-n)** 161

6.6 **five==.mac** 164

6.7 Closeup of == Organelle in **five==.mac** 165

6.8 **five=.mac** 166

6.9 Closeup of = Organelle in **five=.mac** 167

A.1 **five==.mac** With an Incorrect Organelle Specification 172

A.2 **five==.mac** Organelle Detail 173

A.3 **five=.mac** Pins With Correct CIF Scale 175

A.4 **five=.mac** Pins With Erroneous CIF Scale 176

A.5 **five=.mac** Modified Pins 181

ACKNOWLEDGEMENT

Many thanks to Professors D. Kirk, H. Loomis and B. MacLennan for valuable time spent discussing this thesis and related ideas.

Professors R. Hamming and R. Strum, with their iconoclastic manner, were a cheerful source of encouragement.

MIT Lincoln Laboratories provided a copy of LBS and MacPitts to work with. Their research in this area was a source of motivation.

Dr. A. Domic and Dr. D. Johannsen through their lectures, writings and personal contact answered many questions and fostered an interest in programs that make computers.

Special appreciation is extended to B. Limes, D. Shaffer, A. Wong, S. Whalen, M. Williams and R. Johnson for their technical expertise and support.

Dr. F. A. Malagón-Díaz and Mrs. E. Fajar de Malagón provided many years of loving nurture and the belief that ideas shape the world.

CAPT E. Malagón, USMC, read through several drafts and gave many constructive comments.

LT A. Mullarky, USN, created the equality cell which was used to modify a MacPitts' functional unit.

MAJ E. Weist, USMC, worked on a graphical interface to MacPitts and gave useful insights into the compiler's more arcane aspects.

This thesis is dedicated to the American people and Navy.

I. INTRODUCTION

A. BACKGROUND

LISP is often associated with Artificial Intelligence (AI) and its many intriguing possibilities. AI has roots that extend deeply into philosophical thought and perhaps originated with Hobbe's view of thinking as computation. But, can machines think? Perhaps it's just that engineers haven't been able to design them to do it.[1] Or, maybe, the circuits to make such machines are inordinately huge and complex. If this is the case, then computers can be used to automate part of the design process and speed things up. For example, a silicon compiler is a program that takes a high level description of an electronic circuit's behavior or function and outputs a VLSI implementation.

Up to now though, silicon compilers have traded off chip efficiency for design time. Their products don't compete well with handcrafted chips. Can AI techniques remedy this situation? In other words, can the silicon compilers be made "smarter"? It seems that the attribute of intelligence, which is AI's goal, is needed now, in order to create the basis for creating itself.

This is a thorny problem which crops up over and over in current AI projects and which has plagued philosophical thought for thousands of years.

---

[1] Richard Hamming. To balance this out, he adds: "Just because you wish to believe that computers can think does not mean that they can."

Modern philosophers like Dreyfus, Haugeland, Heidegger, Husserl and Wittgenstein take different stances on what constitutes intelligence.[2]

In the meantime, success in war and peace depends on computers. Sensors, controllers and actuators melded into smart machines build cars round the clock or kill at long range. Additionally, computing machines process data used in all phases of decision making. The range of use extends from simple word-processors up to expert consultants.

However, the potential use of computers has only begun to be explored. And, though there have been many impressive results from computer expert systems, they have been limited to specific domains of expertise. Therefore, in order to break through to a new level of processing activity, the Defense Advanced Research Projects Agency (DARPA) launched a major Strategic Computing (SC) program. (DARPA, 1983, pp 1-18)

SC has a goal of creating a widespread machine intelligence technology in the United States. It aims at creating a prototype autonomous land vehicle, a pilot's associate and a battle management system. The SC program is multi-level and addresses issues from microelectronics to software design. However, several areas, such as vision and speech recognition, which humans do so effortlessly, are difficult for machines with present approaches as indicated in this quote (DARPA, 1983, p. 33):

> Recent progress in developing vision for navigation has been severely constrained by lack of adequate computing hardware. Not only are the machines which are now being used too large to be carried by the experimental vehicles, but current machines are far too slow to execute the vision algorithms in real-time . . . .

---

[2] See the bibliography.

It is estimated that 1 trillion von Neumann equivalent computer operations per second are required to perform the vehicle vision task at a level that will satisfy the autonomous vehicle project's long-range objectives. At best, current machines of reasonable cost achieve processing rates below 100 million operations per second. The required factor of 10^6 improvement in speed will have to be achieved through VLSI implementation of massively parallel architectures.

The creation of new software methodologies for parallel computation and a shift of present software structures into VLSI circuits is creating more powerful processing structures. These may or may not lead to thinking machines, but they will certainly change the nature of computation. Paraphrasing Richard Hamming: If you believe computers can't think, you're probably right; but, if you don't do anything about it, you will be left far behind.

Therefore, in order to understand where automatic VLSI design tools can be enhanced in order to accelerate the development of new processors, two existing silicon compilers were investigated.

B. SCOPE OF THIS THESIS INVESTIGATION

The MacPitts silicon compiler has been previously studied at the Naval Postgraduate School mainly from the user's point of view. (Carlson, 1984)(Froede, 1985)(Larrabee, 1985) From these studies two things became evident:

(1) The user interface to MacPitts can be made more accessible to the systems engineer.

(2) Changes to the compiler require a study of the LISP code it is written in.

The first problem is addressed by creating a flowchart interface in which the user graphically creates state diagrams that are converted for the user into MacPitts programs. (Weist, 1986)

The second issue is the subject of this thesis: an examination of Lincoln Laboratory's LISP based Layout Language (L5) and its relation to MacPitts. L5 is a LISP based language used by MacPitts to compile Very Large Scale Integrated (VLSI) circuits automatically. L5 is also used by the Lincoln Boolean Synthesizer (LBS), a Complementary Metal Oxide Semiconductor (CMOS) compiler of arbitrary boolean expressions, to generate combinational logic circuits.

Both of these compilers have many interacting programs linked together to execute automatically. Alteration of this behaviour requires that the programs, composed of L5 and LISP code, be modified.

Therefore, the main questions examined in this thesis are:

- How is L5 created?
- How is L5 used?

The answer to these questions is given by:

- Introducing LISP;
- Covering LISP extensions needed to create L5 (lincoln.l);
- Presenting L5;
- Grouping several programs into a "compiler"; and,
- Modifying a MacPitts functional unit.

LISP fundamentals are covered in Chapter II. The ideas of functional programming and other general concepts are discussed. After this overview, the presentation covers LISP functions and usage. Additionally, a look is

taken at debugging tools available in the Franz Lisp programming environment.

In Chapter III a program that contains many of the basic functions used in LBS and MacPitts, lincoln.l, is presented. The key concept in lincoln.l is the data structure generation macro **defstruct**. This macro is an example of a LISP function that creates other LISP functions. It is the method used throughout both compilers to generate easily manipulated data structures.

From the discussion of extensions made to the LISP language in lincoln.l, Chapter IV moves on to L5. The primitive layout objects, items, and their data structures are shown. The functions which are used in L5 to generate complex structures out of these basic items are illustrated. The interrelationship of L5 with two other hierarchically organized formats, Caesar and CIF, is also covered.

After this, Chapter V deals with linking a group of LISP programs into an environment which the user can run as an integrated system. The examples used are the LBS and MacPitts top-level functions. These functions control program execution.

In Chapter VI a modification to a MacPitts functional unit, an organelle, is described. The material in this chapter covers enough of MacPitts to enable the user to experiment with changing MacPitts' data-path. However, once these basic ideas are understood, then other portions of the compiler can also be changed.

The last chapter, Chapter VII, presents thesis conclusions and contains suggestions for future work.

Appendix A contains a description of alignment problems caused by incorrect CIF plotting or organelle specification; and, a sketch of how to experiment in the MacPitts environment.

In summary, this thesis covers L5, a flexible idiom for procedurally creating VLSI circuits, and shows how understanding L5 makes MacPitts and LBS accessible for modification.

II. THE LISP ENVIRONMENT

LISP is a flexible language with a rich set of programming tools. It can be run interactively or it can be compiled. LISP allows interpreters for other languages to be easily written. For example, both Macpitts' and LBS's interpreters are LISP based. The following quote summarizes some other reasons for learning LISP (Hofstadter, 1985, p. 450):

> Beginners in Lisp encounter . . . fundamental issues in computer science that even some advanced programmers in other languages may not have encountered or thought about. Such concepts as lists, recursion, side effects, quoting and evaluating pieces of code . . . are truly central to the understanding of the potential of computing machinery.

This chapter examines these ideas, shows LISP's applicative [functional] nature and briefly covers LISP's lexicon. The last chapter section is an introduction to useful LISP program development tools.

A. LISP: A FUNCTIONAL PROGRAMMING LANGUAGE

This section covers functional programming and introduces a concise language for talking about program syntax [Backus Naur Format]. A look is taken at LISP's basic functional form [lambda notation] and the issues associated with passing parameters into functions. The problem of variable scoping is also covered. Finally, a brief comparison of iteration and recursion is given.

1. Functional Programming

Imperative languages are based on directing control through a series of assignment statements. LISP on the other hand applies functions to their arguments. (MacLennan, 1983, p. 345)

A function takes a combination of arguments and assigns a unique value to it. A *functional* or *applicative* language[1] is built upon a simple idea that is well illustrated in this quote (Hofstadter, 1985, p. 452):

> A programmer's instinct says that you can cumulatively build a system, encapsulating all the complexity of one layer into a few functions, then building the next layer up by exploiting the efficient and compact functions defined in the preceding layer. This hierarchical mode of buildup would seem to allow you to make arbitrarily complex actions be represented at the top level by very simple function calls.

This spirit of functional application pervades both MacPitts and LBS. But, before looking at LISP's functions, a language for talking about LISP, Backus Naur Format, is introduced.

2. Backus Naur Format (BNF)

BNF is a concise set of symbols for describing the syntax of computer languages. Its key idea is that the description should look like the language it's talking about (MacLennan, 1983, pp. 166-173). A terse set of BNF symbols is given below:

- The " < " and " > " indicate syntactic categories. For example, <integer>, <LISP form>, etc..
- The " ::= " means " is defined as ".

---

[1] Haugeland, 1984, pp. 125-164 gives a very cogent explanation of several computer architectures [LISP included].

- A straight line, " | ", stands for " or ".
- Square brackets, " [" and "] ", indicate optional parameters.
- An asterisk, " * " is equivalent to " a sequence of zero or more "
- A " + " means " a sequence of one or more ".
- Braces, " { " and " } " are used to group syntactic classes and say " a sequence of <class$_1$>s or <class$_2$>s, etc.".
- Many LISP symbols represent themselves. For example the LISP prompt sign " **->** " or LISP parenthesis " **(** " and " **)** "[2]

Consider an example that creates a class of <other character>s.

<other character> ::= **+** | **-** | **_** | **%** | **!** | **?**| **&** | ***** | **@**

In other words, the <other character>s are: **+** or **-** or **_** or **%** or **!**, etc.. Another class is <alphanumeric>s, defined as follows:

<alphanumeric> ::= <letter> | <digit>

Therefore, an <alphanumeric> is either a <letter> or a <digit>. A special LISP object, an **atom**, is defined as a sequence of previously defined objects in the following manner.

<atom> ::= {<other character> | <alphanumeric>}$^+$ | **()**

An **atom** is a sequence of one or more <other character>s or <alphanumeric>s; or, **()**. The empty **atom**, **()**, is called **nil**. [Hasemer, 1984, p. 5] [Refer also to Table 2.1 and Figure 2.2 in Section II.B.1.b] Atoms are basic LISP building blocks.

---

[2] " **->** " is Franz Lisp's prompt sign. " **(** " and " **)** " respectively start and stop LISP's interpretation of an instruction. See Section II.B.1.a for an explanation.

There is another important LISP object, a **list**, defined below:

<list>[3] ::= **(**<atom>*)** | **(**{<atom> | <list>}*)**

A **list** is a left parenthesis followed with zero or more atoms or lists, closed off with a right parenthesis. Notice that this is a recursive definition: a <list> is defined in terms of itself. Examples of lists are:

(), **(a)**, **(a b (c d) e)**.

Note that **()**, **nil**, is both an **atom** and a **list**.

BNF is used throughout this thesis to describe LISP syntax. LISP's basic functional format, <lambda function>, can now be analyzed.

3. Lambda Functions

One method for writing functions in LISP is with lambda notation. [For other function definition formats see Section II.C.2] Perhaps the easiest way to understand lambda notation is with this quote showing its history (Touretzky, 1984, p. 86):

> Lambda notation was created by Alonzo Church, a mathematician at Princeton University, as an unambiguous way of specifying functions, their inputs, and the computations they perform. In lambda notation, a function that added 3 to a number would be written λx.(3 + x). The λ is the Greek letter lambda.

---

[3] Refer to Sections II.C.1 and II.C.3.b. A list can also be viewed in this light:

<list> ::= **(**<head><tail>**)**
<head> ::= { <atom> | <list> }
<tail> ::= <list>

For example:

()	has <head> := **nil**	and <tail> := **nil**
(a)	has <head> := **a**	and <tail> := **nil**
(a b (c d) e)	has <head> := **a**	and <tail> := **(b (c d) e)**

John McCarthy, the originator of Lisp, adopted Church's notation for specifying functions. The Lisp equivalent of the unnamed function $\lambda x.(3 + x)$ is the list

(LAMBDA (X) (PLUS 3 X))

A function $F(x,y) = 3x + y^2$ would be written $\lambda(x,y).(3x + y^2)$ in lambda notation. In Lisp it is written

(LAMBDA (X Y)(PLUS (TIMES 3 X) (TIMES Y Y)))

Lambda notation creates functions in LISP with this syntax:

<lambda function> ::=
(lambda (<argument>*****)<LISP form>**)**
<LISP form> ::= {<atom> | <list>}+
<argument> ::= <atom>

A lambda function is used [*applied* to parameters] to obtain a value with this format:

<value> ::= **->** **(**<lambda function><parameter>*****)<CR>[4]
<value> ::= <LISP form>
<parameter> ::= <LISP form>

Therefore, to apply the function $F(x,y) = 3x + y^2$ with x=2 and y=3 in LISP, the user types:

-> ((lambda (x y)(plus (times 3 x)(times y y))) 2 3)<CR>
;; ((lambda (x y)(plus (times 3 x)(times y y))) <x> <y>)
;;[5] This function is equivalent to:
;; (+ (* 3 2)(* 3 3)) = (+ 6 9) = 15
15

That is to say, that the value resulting from F(2,3) is equal to 15.

---

[4] <CR> is an abbreviation for the action of hitting a "carriage return"

[5] LISP ignores anything on a line after a semicolon; therefore, one or more semicolons " ; " are used to insert comments into LISP programs.

The lambda function format can be named by using the LISP primitive **def**[6] in this manner:

<function-name> ::=
-> (def <function-name> <lambda function>**)** <CR>
<function-name> ::= <atom>

A function created with **def** is applied to its argument's parameters by using its name as follows:

<value> ::= **->** **(**<function-name><parameter>***)**<CR>

By naming the function, its usefulness is increased. Instead of typing the unwieldy lambda form each time the function is applied, the user simply types in the function's name. Consider F(x,y) = 3x + y2 defined as a LISP function named **quadratic**:

```
-> (def quadratic
;; <function-name> ::= -> (def <function-name><lambda function>)
      (lambda (x y)
        (plus (times 3 x)(times y y))))<CR>
;; LISP returns <function-name>:
    quadratic
```

This function, **quadratic**, is applied by using its name with parameters:

```
-> (quadratic 2 3)<CR>
;; (quadratic <x><y>)
        15

-> (quadratic (quadratic -1 2)(quadratic 2 3))<CR>
;; (quadratic -1 2) := 1 & (quadratic 2 3) := 15
;; (quadratic 1 15) := 228
        228
```

---

[6] See Section II.C.2 for another method for defining functions [**defun**].

The following question now arises: If functions are nested and values passed from function to function, then how are the values of these variables being controlled? That is to say, the above computation assumed this structure:

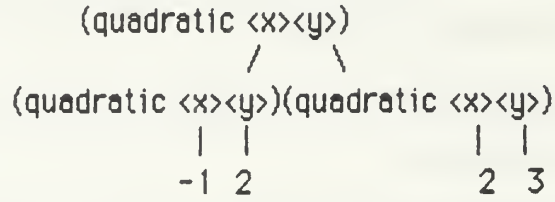

Figure 2.1 Passing of Values in Nested Functions

But, why not some other method? The next section examines established conventions for passing values and calling variables.

4. Variable Scoping

The context in which a variable is called can in fact affect its value. The method used to call up a variable and limit its scope will determine the value that is used. To examine these ideas a few definitions are needed (Rosenberg, 1984, pp. 62, 165, 460, 501, 570):

call by reference: a subroutine or procedure call where the addresses of the parameter's storage locations are passed to the subroutine.

call by value: a subroutine or procedure call where the actual values of the parameters are passed to the subroutine.

dynamic: occurring at the time of execution.

scope (of a variable): the portion of a computer program within which the definition of the variable remains unchanged.

static variable: a variable that is allocated before execution of the program begins and that remains allocated for the duration of execution of the program.

variable (V)(VAR)

(1) in computer programming, a character or group of characters that refers to a value and, in the execution of a computer program, corresponds to an address.

(2) a quantity which can assume any of a given set of values . . . .

Three more terms need to be defined: A *bound variable* is one of a function's formal parameters [function's arguments]. A *global variable* has its value set at the top level. A *free variable* is not a bound variable, but its value is used or changed by a function. (Wilensky, 1984, pp. 39-40) Now that the terms have been defined, the concept of variable scoping can be examined.

There are two basic variable scoping techniques "-- static scoping and dynamic scoping. In static scoping (also called lexical scoping) a procedure is called in the environment of its definition; in dynamic scoping a procedure is called in the environment of its caller." (MacLennan, pp. 112-113, 1983). In other words, (MacLennan, p. 109, 1983):

- In dynamic scoping the meanings of statements and expressions are determined by the dynamic structure of the computations evolving in time.

- In static scoping the meanings of statements and expressions are determined by the static structure of the program.

Franz LISP is a dynamically scoped language.[7] Therefore, bound variables which are changed during a function call are restored to their original values upon exiting the function. If calls to other functions are

---

[7] COMMON LISP is a lexically scoped language (Winston, 1984, p. 54).

called function. However, changes to a free variable are not restored. If the free variable has the same name as a global variable, then other instances of the global variable will also change! (Wilensky, 1984, pp. 40-41)(Winston, 1984, pp. 54-55)(MacLennan, 1983, pp. 284-288)

An additional consideration, besides variable scoping, is the method used to pass parameters to a function. This is done in either of two ways: *call by value* or *call by reference*. LISP uses *call by value*.

To explain the difference in the two methods, recall that functions have a list of arguments [formal parameters or dummy variables]. These arguments correspond to a list of actual parameters when the function is applied. In other words, the arguments are bound to the actual parameters. In most programming languages the user can assign values to symbols and these symbols can be used as actual parameters in a function. The issue now becomes one of passing the symbol's value or address. In *call by reference* the address of the actual parameter is bound to the argument. *Call by value* only copies the actual parameter's value: control over the actual parameter's value is not handed over. (MacLennan, 1983, pp. 53-58)

The difference in the two methods can be seen by examining the effect of using a free variable, "free", whose value has been set at 2 (Winston, 1984, p. 55):

```
-> (setq[8] free 2)<CR>
;; The global variable "free" is set to a value of 2, e.g.,
;; free := 2
        2
```

---

[8] See Section II.C.3.a for methods of binding symbols in LISP.

```
-> (defun[9] test (bound)
       (setq bound (1+ bound))
;; bound := bound + 1
;; The symbol "free" is not bound within the context of test.
       (+ bound free))<CR>
;; the result := bound + free
         test

-> (test free)<CR>
;; First, "bound" assumes "free's" value: bound := free's value := 2
;; Second, bound := bound + 1 := 2 + 1 := 3
;; Third, the result := bound + free := 3 +2  = 5
         5
```

In contrast if LISP used *call by reference* :

(1) bound ::= free := 2

(2) "bound" increments: bound := bound + 1 := 1 + 2 = 3

(3) since bound ::= free, "free" also becomes 3: free := 3

(4) the result is: bound + free := 3 + 3 = 6

In summary, Franz Lisp resolves the problems of variable context and scoping by using call by value and dynamic scoping. This issue can be extended to functions. Next, consider how functions refer to other functions or to themselves.

5. Recursion and Iteration

LISP allows functions to refer to themselves. This approach, known as recursion, is briefly introduced in this section.[10] Suppose a function that raises a given integer base to a nonnegative integer power is desired. Two

---

[9] **defun** is an alternate method of defining functions, see Section II.C.2.

[10] A more in depth discussion of recursion is given in Section II.C.4.

possible algorithmic approaches are:

(1) Iterative:

(a) Set the result to 1;

(b) set the index to the power;

(c) iterate, by multiplying the result with the base and reducing the index by 1; and,

(d) stop when the index reaches zero.

The iterative algorithm divides up an operation into increments which are repeated a specified number of times. In recursion, a function calls itself until a basis condition is reached. The key to writing recursive functions is to ensure the basis condition is well formulated, for example:

(2) Recursive:

(a) If the power is zero give a result of 1. [BASIS] Otherwise,

(b) multiply the base with the result of applying this algorithm to the original base and the power reduced by one. [RECURSION]

In an imperative language iteration is commonly used, whereas in an applicative language recursion often seems more natural (Gray, 1984, p. 51). The above example shows how an iterative problem can be stated recursively. But, although iteration and recursion are theoretically equivalent, it's not always trivial to convert from one to the other. (MacLennan, 1983, p. 394)

More will be said about iteration and recursion in Section II.C, in the meantime, a discussion of how Franz Lisp is used is now presented.

B. INTERPRETED, COMPILED OR DUMPED LISP

The interpreter allows interactive running of LISP programs and provides an effective environment for debugging LISP code. At the same time, LISP also provides a compiler which can considerably speed up program execution for large code segments. This section examines the different ways LISP can be run and covers very basic input and output.

1. The LISP Read-Eval-Print Loop: Interpreted LISP

In Section II.A.1, several examples showed how the LISP interpreter "reads" and "evaluates" input, and then "prints" out a result. This read-eval-print loop is discussed in this section. The two major participants in this cycle, **eval** and **quote**, are also covered.

a. The LISP Prompt " -> ", Start " (" and Stop ") "

To obtain " -> ", so that <LISP form>s with " (" and ") " can run, the Franz Lisp interpreter is invoked by typing **lisp** after the UNIX® prompt:

```
% lisp<CR>
Franz Lisp Opus 38.69
->
```

The " -> " is a prompt sign which means that inputs will be "evaluated" or "interpreted". An open parenthesis, " (", instructs the interpreter to do whatever follows, and a closed parenthesis, ") ", tells the interpreter to stop doing it. (Wilensky, 1984, p. 2)(Hasemer, 1984, p. 6) Therefore, if the user inputs: **(plus 1 2 3)** <CR>, the " (" starts the LISP interpreter "plusing" **1** with **2**, then with **3**, and stops "plusing" upon reaching ") ". For example:

```
-> (plus 1 2 3) <CR>
     6
```

This cycle of reading inputs, evaluating them and returning a result is referred to as the LISP read-eval-print loop. (Hasemer, 1984, p.4) But what does it mean to say that the interpreter "evaluates" <LISP form>s? The next section answers this question by describing the function used by the LISP system to obtain the results of a <LISP form> input after the prompt sign.

b. A Universal LISP Function: **eval**[11]

When the interpreter "reads" input, it is applying a function, **eval**, which can interpret all LISP functions. The evaluation function's syntax is illustrated below:

<value> ::= -> **(eval** <LISP form>**)**
<value> ::= <LISP form>

The **eval** function operates as follows (Winston, 1984, p 34):

(1) If the form is a <number>: return the <number>. Otherwise,

(2) If the form is a <symbol>: return its <value>. Otherwise,

(3) If there is a <special symbol>: it's an exception. Otherwise,

(4) Apply **eval** to the <list>'s tail,[12] then

(5) Apply the <list>'s head to the <list>'s evaluated tail.

Eval assumes the hierarchy of LISP objects (Winston, 1984, p.21) shown in Figure 2.3 below:

---

[11]When John McCarthy was developing LISP he proved that there is a universal LISP function, **eval**, which can interpret any LISP function. This is similar to the universal Turing machine that can simulate any other Turing machine. (Charniak, 1985, p. 48)(MacLennan, 1983, p. 343)

[12] Refer to Sections II.A.2, II.C.1 and II.C.3.b.

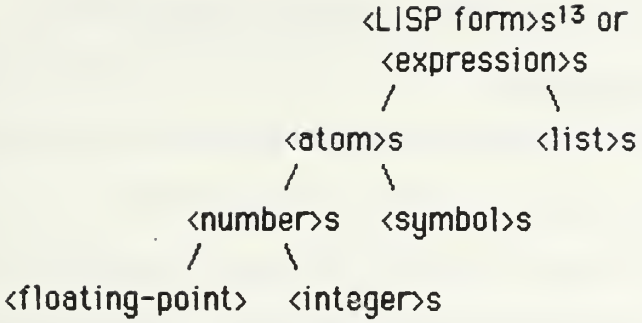

Figure 2.2 The LISP Object Hierarchy

Examples of these LISP objects are shown in Table 2.1:

TABLE 2.1

EXAMPLES OF LISP OBJECTS

LISP Objects	Example LISP Code
• <LISP form>s	**(plus 1 2)**, **1.23**, **(* (plus 1 2) 3)**, . . .
• <list>s	**()**, **(((q w)(e) 1 2))**, **(plus 1 2)**, . . .
• <atom>s	**1**, **1.1**, **A**, **woman**, . . .
• <symbol>s	**a**, **man**, **wo223**, **%%err**, . . .
• <special symbol>s	**" ' "**, **" \ "**, **" , "**, **" ; "**, **" ,@ "**[14] . . .

It seems that LISP is always searching for a value. The next section answers the question: "How does it accept something literally?"

c. Eval's dual: **quote** or " ' "

When evaluation is undesirable it is inhibited with **quote** or its abbreviated form, a quote mark. The " ' " is a <special symbol> that stops evaluation. This idea is evident from the syntax:

<LISP form> ::= -> {**(quote** <LISP form>**)** | '<LISP form>}<CR>

---

[13] Refer to Sections II.A.2, II.A.3 and II.C.1.

[14] "\" [backslash] is an escape character. " ," and " ,@ " are described in Section III.A.1.2

A few examples illustrate the effect of **quote** or " ' ":

-> (quote (a b))<CR>
;; (quote <LISP form>)
(a b)

-> '(a b)<CR>
(a b)

-> (a b)<CR>
;; If "(a b)" is input without first defining "a" as a function, then
;; an error results. The error is cleared using the reset function.
Error: eval: Undefined function a
<1>: (reset)<CR>

Two examples now show how the quote mark is used and the error that results when a variable is evaluated without having been given a value:

-> 'australanthropus<CR>
australanthropus

-> a<CR>
Error: Undefined variable: a
<1>:

The variable "a" doesn't have a value; therefore, the LISP system complains with an error message, leaves the standard read-eval-print loop and enters a debugging loop. The " **<1>:** " is the LISP error prompt. The user can continue typing expressions. Or, to remove the error, simply reset the system as follows (Wilensky, 1984, p.6):

<1>: (reset)<CR>
[Return to top level]
->

However, there are errors where execution might not be stopped by the interpreter[15]. In that case, LISP can be stopped with an interrupt. The first control C [^C] sets an interrupt flag: the system waits for a "safe" place to exit. The second ^C forces all system calls to compiled code to check the interrupt flag; and finally, a third ^C causes an immediate interrupt. (Foderado, 1983, Section 10.6) Here is an example:

```
^C ^C ^C
Interrupt: ^C
Break nil
<1>:
```

An interpreter is a useful interactive tool; however, to handle large programs and obtain efficient object code, a compiler is needed.

2. Compiled LISP

Compilation of LISP programs increases their execution speed. In order to keep compilation dependencies among several programs straightened out, a makefile is used. In addition, a makefile can join together several programs so they can run as a large unit.

a. The Compiler

The Franz Lisp compiler is invoked from the UNIX® C-Shell with the following command (Foderado, 1983, Chapter 12):

% liszt [-<option>*] <filename>

There are several options, among which, **-q** [compile in quiet mode] and **-x** [create a cross reference file] are very useful. The compiler can be run with several options at one time as follows:

---

[15] Richard Hamming has jokingly said that perhaps computers do in fact show free will, it's just that people always call a repairman when they do it.

% liszt -qx lincoln.l[16]<CR>

This command will create a lincoln.o [object code file] and lincoln.x [cross reference file]. To use the cross reference file, it must be filtered through the program lxref:

% lxref lincoln.x > lincoln-x-ref<CR>

The " > " is used in UNIX® to redirect output, in this case, the human readable cross reference file is placed in lincoln-x-ref. This file will show all the functions in lincoln and where they are used.

If functions from one program are used in another, this can then present a problem during compilation. How can these interrelationships be effectively managed? A makefile is the answer.

b. Compilation Dependencies: Makefile

Large programs such as LBS or MacPitts are subdivided into functional parts; however, between these subprograms there exist dependencies. For example, lincoln.l contains primitive functions used by L5.l [the layout language] and L5.l is in turn used to describe basic circuit layouts in organelles.l [functional units]. If there are twenty subprograms each with dependencies, then making changes can become a major chore. UNIX® has a utility to ease this difficulty: makefile. A makefile consists of dependencies, prerequisites and actions with this syntax:

<makefile> ::= {<dependent result> : <prerequisite result>*
<action>}+

---

[16] linconl.l is a file containing many useful lisp functions used at MIT Lincoln Laboratories, see Chapter III.

Consider the following example makefile composed of four dependent results [**L5.o**, **work**, **clean**, and **doc**]. The desired results are separated by a colon from their prerequisites and placed on the same line. Notice that two results, **clean**, and **doc**, have no associated prerequisites. The next line contains the actions to create each result. Assume that all of this code is in a file named " Makefile " in the user's directory which contains L5.l and lincoln.l. Makefile's contents are now presented, and described immediately afterwards [the explanation continues into Section II.B.3:

```
L5.o: L5.l lincoln.o
    liszt -qx L5

work: L5.o lincoln.o
    echo[17] "(eval-when (eval)\[18]
        (load 'lincoln.o)(load 'L5.o)\
        (dumplisp[19] work)(exit))" | lisp

clean: rm[20] -f L5.o lincoln.o
```

---

[17] The **echo** command prints out its arguments. The function, **eval-when**, tells the LISP compiler to evaluate the expressions that follow, instead of compiling them. (Wilensky, 1983, 281)

[18] The backslash " \ " is an escape character, therefore the next line is treated as a continuation. The " | " stands for "pipe", i.e., the results of the first process are passed on to the next process.

[19] Saves the LISP environment in an executable file named "work". Typing "work" will then recreate the LISP system as it was running when it was dumped.

[20] Forced removal of files.

```
doc:
    echo " print  L5.l lincoln.l Makefile " | csh
```

This makefile can compile L5.1, and ensure that the dependent files are updated before doing it, by using the UNIX® **make** command:

```
% make L5.o<CR>
# The <prerequisite result> is lincoln.o, so this is created first:
/usr/ucb/liszt -qx lincoln
# The liszt compiler takes the lincoln source file and compiles
# it; however, there are several warnings that can be ignored.
%Warning: lincoln.l : rotate : . . .
# Now that all the prerequisites are complete, L5.o is compiled.
%/usr/ucb/liszt -qx L5
# When everything is done the UNIX® system is restored.
%
```

In a similar fashion to print out the makefile and source code files:

```
% make doc<CR>
#21 Pipe the print organelles.l etc. command to the UNIX® shell.
# This causes these files to be printed out.
    echo " L5.l lincoln.l Makefile" | csh
# Notice that the <action>s are printed out. In this case the user
# would go to the line printer and print up copies of these files.
%
```

Or, perhaps, to remove undesired object code files and list the files:

```
% make clean<CR>
# Remove files ending in ".o" or ".x" and then list the directory
# contents on the terminal.
    rm -f *.o
    rm -f *.x
    ls
# The ".o" and ".x" files are removed and a directory listing given:
    DIR             READ_ME_FIRST       chip.l
```

---

21 The UNIX® shell ignores anything after " # ", therefore this symbol is used to insert comments.

```
Examples        array.l         top.l
INSTALL         bin             extract.l
L5.l            c-routines.c    sim.l
Makefile
```

* And finally, back to the UNIX® prompt.

```
%
```

LISP files which are dependent on each other can be organized using a makefile. They can also be individually loaded into the interpreter and saved as one executable file using **dumplisp**.

3. Interpreted and Compiled LISP: **dumplisp**

In some programming languages disparate programs can be combined to form a working unit using a linker. In LISP this can be achieved by creating an "environment" that contains all the programs. This is what the **work** section of the example makefile created in the previous section does:

```
% make work<CR>
* Execute the actions under the "work" heading of this makefile.
    echo "(eval-when (eval)(load 'lincoln.o)(load 'L5.0)
        (dumplisp work)(exit))" | lisp
* Load lincoln.o, L5.o and organelles.o into LISP, dump this envi-
*ronment in an executable file named "work" and then exit LISP.
    Franz Lisp, Opus 38.69
-> [fasl lincoln.o]
;; fasl is the function LISP uses to load object code files.
-> [fasl L5.o]
%
```

In summmary, an executable file, **work**, has been created. Typing **work** as an imperative command places the user in LISP with the functions in L5 and lincoln also available.

```
% work<CR>
->
```

This is a LISP environment that contains lincoln and L5. If the user wants to load another file into this environment the **load**[22] function can be used:

```
-> (load 'defstructs.o)<CR>
;; (load '<filename>)
;; The function LISP uses to load object code is fasl.
[fasl defstructs.o]
    t

-> (load 'top.l)<CR>
;; Source code can also be loaded into the interpreter.
[load top.l]
    t
```

To save all these changes in the LISP environment:

```
-> (dumplisp 'temporary)<CR>
;; Dumping LISP into "work" will result in an error since work is
;; the process that's running LISP.
    nil
```

To leave LISP and return to UNIX® :

```
-> (exit)<CR>

% mv temporary work<CR>
* "mv" moves the file "temporary" into the file "work".
```

Another approach for creating a LISP environment is to use a .lisprc file. The LISP interpreter always checks the user's directory to see if a .lisprc file with instructions exists. An example .lisprc file that loads lincoln.o, L5.o and organelles.o into LISP is:

---

[22] The function **include** also places files into LISP. Unlike **load**, it does not evaluate its argument.[Foderado, 1983, p. 6-4][Wilensky, 1984, p. 282]

```
% cat .lisprc<CR>
* The UNIX® "cat" command dumps the file ".lisprc" onto the
* terminal screen.
    (eval-when (load eval)
        (load 'lincoln.o)
        (load 'L5.o)
        (load 'organelles.o) )
```

Since LISP automatically loads the .lisprc file [in this case all that the file contains is one large **eval-when** <LISP form>], then the result is that all three **load** functions are evaluated and the files loaded in.

```
% lisp<CR>
* The lisp interpreter is invoked and the .lisprc file is loaded.
Franz Lisp Opus 38.69
->
```

The user is now in LISP with the three files loaded. The main difference between using this method and **dumplisp** is that a dumped file usually requires at least a megabyte of storage, whereas loading several files using the .lisprc file takes a short while.[23] In Chapter V.A and Appendix A.B it will be seen that the MacPitts and LBS environments can be invoked by typing their respective names without any arguments. For example:

```
% macpitts [or lbs]
usage: macpitts <filename> [<options>]
->
```

A closer look is now taken at how files are input into LISP and how functions can be output into files.

---

[23] A compromise between these two approaches is to use the autorun option when compiling a LISP file [e.g., **% liszt -r** <filename>]. This creates an object file which has a small piece of bootstrap code attached. The object file can then be run as an executable file. (Wilensky, 1984, p. 284)

4. Basic Input and Output [I/O]

Large programs are usually stored in files and loaded into LISP using **load**.[24] They can be edited using the vi editor by calling the LISP functions **vi** or **vil**:

-> (vi[25] top.l)<CR>
;; Places the user in VI editing the file top.l and when the user
;; finishes returns back to LISP.
t

-> (vil organelles.l)<CR>
;; Places organelles.l into VI and when editing is complete, loads
;; organelles.l into LISP and returns nil.
[load organelles.l]
nil

Functions that have been created in the interpreter can be output into a file using the pretty print, **pp**, function. (Foderado, 1983, Chapter 5) (Wilensky, 1984, pp 134-137) Part of this function's syntax is shown below:

<LISP form> ::=
-> (pp [(F <file-name>**)]** {<function-name> | <symbol>**)**}<CR>

For example, to pretty print out the **m-to-the-n** function:

-> (pp m-to-the-n)<CR>
;; Output the function m-to-the-n to the screen.

```
(def m-to-the-n
     (lambda (m n)
       (cond ((zerop n) 1)
             (t (times m (m-to-the-n m (1- n))))) ) ) )
```

---

[24] See the previous discussion on interpreted LISP.

[25] The MacPitts version of L5 has a **vi** function defined in it. However, it automatically loads the file into LISP.

The pretty print function can also be used to send <LISP form>s to a file in the following fashion:

-> (pp[26] (F temp.l) m-to-the-n)<CR>
;; Output the function m-to-the-n to the file temp.l.
t

Conversely, a <LISP form> can be read from a file, without being evaluated, using **read**:

-> (read (infile 'temp.l))<CR>
;; Read the next <LISP form> from the temp.l file. When the end of
;; file is reached then nil is returned. The <LISP form> is not
;; evaluated when read. To do so eval must be explicitly used. For
;; example: (eval (read (infile '4-flags))), where 4-flags has a
;; <LISP form> that needs to be evaluated.

```
(def m-to-the-n
     (lambda (m n)
       (cond ((zerop n) 1)
             (t (times m (m-to-the-n m (1- n))))) ) ) )
```

-> (exit)<CR>
;; Leave LISP and then output temp.l to the screen using cat

---

[26] Other functions that are used for output are **patom** and **print**. Their syntax is similar:

<LISP form> ::=
-> (patom [']<LISP form> [**(outfile** <filename> ['**a**])])<CR>
<LISP form>::=
-> (print [']<LISP form> [**(outfile** <filename> ['**a**])])<CR>

These functions both output to the terminal if the optional outfile argument is not given [the '**a** appends the output to the previous file contents, otherwise they are wiped out]. Because these functions do not send carriage returns when they finish their output, they are usually seen in conjunction with **(terpri** [**(outfile** <filename> ['**a**]) which outputs a terminate line character sequence. For example:

-> (patom '| Stop printing. |)(terpri)<CR>
Stop printing.

```
% cat temp.l<CR>
* The contents of temp.l are concatenated to the screen.
   (def m-to-the-n
    (lambda (m n)
      (cond ((zerop n) 1)
            (t (times m (m-to-the-n m (1- n)))) ) ) )
```

Now, reentering LISP, the temp.l file is loaded into LISP and the function it contains is tested in the following sequence of events:

```
% lisp<CR>
;; Start the LISP environment.
Franz Lisp, Opus 38.69

-> (include temp.l)<CR>
;; Load the temp.l file into which has the m-to-the-n function.
[load temp.l]
    t

-> (m-to-the-n 4 3)<CR>
;; A test of the m-to-the-n function: 4^3 := 64.
       64
```

There are many other I/O functions in LISP, which give the user a great deal of control, but these can be added to this basic set as the user's needs grow. The next section presents a brief LISP lexicon and covers iteration and recursion.

C. LISP FUNCTIONS AND DATA

Part of LISP's power is that it has the same format for data and functions; thereby, allowing functions to be manipulated as data. This idea is elaborated in this section along with an explanation of basic LISP primitives and control structures. These basic LISP constructs allow iterative or recursive algorithms to be easily implemented.

1. LISP's Basic Structure: The List

A function and a list of data look the same in LISP. For example, the next <LISP form>,

(replace-item-points inverter new-points),

is an application of a function [**replace-item-points**] to its arguments [**inverter** and **new-points**]; or, it can also be a list of three elements [**replace-item-points**, **inverter** and **new-points**].

Which one it it? It is both! A LISP program is a list, and **eval** normally applies the list's head as a function to the list's tail. If the list is quoted, then it's treated as data. (MacLennan, 1983, p. 348)

Atoms and lists are referred to as symbolic *expressions*. Expressions are called *forms* if they are to be evaluated. "Considered as data, a list may be called an expression; considered as a piece of procedure, the same list may be called a form". (Winston, 1984, p. 20)

With these ideas in mind another look can be taken at the procedure for LISP function definition.

2. LISP Function Definition: **def** and **defun**[27]

Up to this point the reader has seen functions that take a fixed number of arguments all of which are evaluated. This class of functions is called an *expr*. There are three other categories: *fexpr*, *lexpr* and *macros*[28]. An fexpr takes an unlimited number of arguments, but does'nt evaluate them.

---

[27] See Section II.A.2 for function definition using **def**.

[28] Macros are discussed in Chapter III.

The lexpr accepts a variable number of arguments and evaluates them. (Wilensky, pp. 116-122, 160-178)

Two ways of creating expr's have been shown: **def** with a lambda function, or a lambda function by itself. There is another syntax for defining an expr [**def**ine **fun**ction] which, unlike **def**, doesn't use lambda notation:

(defun <function-name>**(**<argument>***)** <LISP form>***)**

This function is applied in the same fashion as before:

<value> ::= **->** **(**<function-name><parameter>***)**<CR>

Therefore, in summary, there are three ways of creating expr's:

<expr> ::= **(lambda (**<argument>***)**<LISP form>***)**

<expr> ::= **(def** <function-name>
(lambda (<argument>***)**<LISP form>***))**

<expr> ::= **(defun** <function-name>**(**<argument>***)** <LISP form>***)**

It is sometimes desirable to have a variable number of evaluated arguments in a function. There are several formats for lexpr's:

<lexpr> ::= **(defun** <function-name>
(<argument>* **&optional**[29] <optional-argument>***)**
<LISP form>***)**

<lexpr> ::= **(defun** <function-name><symbol><LISP form>***)**

<lexpr> ::= **(def** <function-name>
(lexpr (<symbol>**)** <LISP form>***))**

---

[29] See Section II.C.3.f for another example of the **&optional** feature in the function **match-that** [Foderado, 1983, p. 4-4].

For example, a function that finds the logarithm base 2 of a number can be defined in LISP as follows:

```
-> (defun log-two (number &optional (base 2))
;; The primitive LISP function quotient finds the quotient of two
;; numbers, and log finds the natural logarithm of a number. The
;; optional argument "base" defaults to a value of 2 if a
;; parameter is not given for it.
    (quotient (log number)(log base)) )<CR>
;; Find the logarithm base 2 or the given base of a number.
        log-two
```

This function is applied in the following ways:

```
-> (log-two 13)<CR>
;; (log-two <number>)
;; Find the log base two [defaul] of 13.
    3.700439718141092

-> (log-two 13 10)<CR>
;; Evaluate the base ten log of 13.
    1.113943352306837
```

Another way to define this lexpr is as follows:

```
-> (defun log-two n
;; In this format, the symbol "n", will be bound with the number
;; of arguments supplied. The function arg gives the parameter
;; associated with the position corresponding to the number it is
;; given.
    (quotient
        (log (arg 1))
;; If a second parameter is provided use its value, if not use 2.
        (log (cond
                ((> n 1)(arg 2))
                (t 2) ) ) ) )<CR>
 log-two
```

The third functional class, an fexpr, doesn't evaluate its arguments and takes a variable number of them. Nothing comes for free though, the flexibility of a variable number of inputs is offset by the overhead of

accessing the parameters. When parameters are input to an fexpr they are all bundled up into a list which is bound to the fexpr's single argument. The parameters must be obtained from the list.

There are two ways to create fexpr's:

<fexpr> ::= **(defun** <function-name> **fexpr**
(<argument>**)**<LISP form>***)**

<fexpr> ::= **(def** <function-name>
(nlambda (<argument>**)**<LISP form>***))**

A good example of an fexpr is a function that loads any number of files without having to quote the files (Wilenksky, 1984, p. 163):

-> (defun load-files fexpr (files)(mapc 'load files))<CR>
;; mapc is similar to mapcar[30] in that it applies a function across
;; one or more lists; however, it doesn't return a useful value, i.e.,
;; the side-effect is what is desired in this case.
load-files

This function can load several files without having to quote them:

-> (load-files lincoln.l L5.l)<CR>
[load lincoln.l]
[load L5.l]
->

These three categories of LISP functions (expr, fexpr and lexpr) are found in different areas of MacPitts and LBS.[31] Some of these applications are examined in later chapters. However, it is useful at this juncture to look at LISP's built-in functions

---

[30] See Section II.C.3.e.

[31] See Chapter IV.

3. Frequently Used LISP Functions

A synopsis of common LISP functions is presented to briefly familiarize the reader with LISP's syntax. First, a look at functions used to give values to symbols.

a. Binding Variables: **set**, **setq**, **let** and **let***

Variables are assigned values with **set** or **setq** [**set** quote]. Although **set** only takes one symbol at a time, it has a similar syntax to **setq**:

(set {['](symbol>} {['](<LISP form>}**)**
(setq {<symbol> ['](<LISP form>}+**)**

These two functions are applied as follows:

```
-> (set 'A '(a b c))<CR>
;; Set "A" to have the value "(a b c)".
    (a b c)

-> A<CR>
;;  A's value is (a b c).
    (a b c)

-> (setq B A  C '(1 2 3) D (plus 1 2 3))<CR>
;; The <symbol>s are unevaluated, but are respectively assigned
;; the results of evaluating the <LISP form>s. setq returns the
;; value of the last evaluation it performs.
;; B := A, C := (1 2 3) and D := (plus 1 2 3) := 6
    6

-> B<CR>
;; B's value has been set to A, but A := (a b c).
    (a b c)

-> C<CR>
;; C's value is (1 2 3).
    (1 2 3)
```

```
-> D<CR>
;; D's value is (plus 1 2 3) := 6
    6
```

let and **let***[32] are used to create an orderly environment in which to assign values to variables, apply functions to the variables and then restore the variables to their original values. Their syntax is similar:

(let[*] ({(<symbol> [']<LISP form>**)}+)**<LISP form>***)**

So, assuming that A, B, C and D still have the above values assigned to them, an example that uses **let** is:

```
-> (let ((A (times 2 3))(B (plus 1 2 3 4))) (list A B) )<CR>
;; First, A and B are assigned values as follows:
;; A := (times 2 3) := 6 and B := (plus 1 2 3 4) := 10
;; Then, the remaining <LISP form> is evaluated as follows:
;; (list A B) := (6 10)
    (6 10)

-> A<CR>
;; Variables are restored to their previous values:
    (a b c)
```

let assigns values in parallel, **let*** does it serially:

```
-> (let*
    ;; First, set A = (+ 1 2 3) = 6
     ((A (plus 1 2 3))
    ;; Second, set B = (* A 5) = (* 6 5) = 30
      (B (times A 5))
    ;; Third, set C = (- B A) = (- 30 6) = 24
      (C (minus B A)) )
     (list A B C) )<CR>
;; The result is a list composed of A, B and C
    (6 30 24)
```

---

[32] **let*** might have to be used if the " * " is not being recognized by the interpreter or compiler. The " \ " serves as an "escape" character.

The variables are restored to the values they had prior to participating in the **let*** construct. With these methods of variable assignment in hand, a look is now taken at list manipulation.

b. List Selection: **car** ,**cdr**[33], **nth** , and **nthcdr**

LISP is based on the application of functions to arguments. The syntax of LISP generally has a structure of the form:

(<function-name><argument>***)**

Therefore, it seems natural to have a selector that picks the first element of a list, the "function", and another selector that returns all the elements of a list except the first, the "arguments". These selectors are **car** and **cdr**:

<head> ::= **->** **(**car <list>**)**<CR>
<tail> ::= **->** **(**cdr <list>**)**<CR>
<list> ::= **(**<head><tail>**)**[34]
<head> ::= <LISP form>
<tail> ::= <LISP form>

The application of these basic selector functions is shown below:

-> (car '(plus 1 2 3 4))<CR>
;; (car <list>)
;; car selects the first ("function" or "head") list element
plus

The " tail" selector, **cdr**, is used as follows:

---

[33] **car** and **cdr** were assembly language instructions for the IBM 704 on which LISP was first implemented. An instruction was divided up into fields. Two of the fields were named the *address* and *decrement*. **car** and **cdr** were the instructions for getting the contents of the address pointed to by these fields. (Charniak, 1985, p.48)

[34] Compare to the definition of a list in Section II.A.2.

-> (cdr '(plus 1 2 3 4))<CR>
;; (cdr <list>)
;; cdr selects all elements except the first ("arguments" or "tail")
;; and returns them as a list
(1 2 3 4)

A more complicated example:

-> (car (car (cdr (cdr '(plus 1 (times 2 3) 4 5)))))<CR>
;; In order to simplify the notation, when car and cdr are applied
;; in succession they are joined into one word, e.g.
;; (car (car (cdr (cdr x)))) would become (caaddr x)
;; cdr : (1 (times 2 3) 4 5)
;; cddr : ((times 2 3) 4 5)
;; caddr : (times 2 3)
;; caaddr : times
times

The next illustration of these selectors uses as an abbreviated format:

-> (cadadddr '(plus 1 (times 2 3) (minus (divide 4 5) 6) 7 8))<CR>
;; cdr : (1 (times 2 3)(minus (divide 4 5) 6) 7 8)
;; cddr : ((times 2 3)(minus (divide 4 5) 6) 7 8)
;; cdddr : ((minus (divide 4 5) 6) 7 8)
;; cadddr : (minus (divide 4 5) 6)
;; cdadddr : ((divide 4 5) 6)
;; cadadddr : (divide 4 5)
(divide 4 5)

There are two other useful list accessors: **nth** and **nthcdr**. They both have very similar syntax:

(nth <number><list>**)**
(nthcdr <number><list>**)**

They are practical alternatives to a succession of **car**s and **cdr**s, and are used in the following manner:

-> (nth 3 '(and in those days it came to pass))<CR>
;; (nth <index><list>)
;; Starting at 0, return the indexed argument of the given list.
days

->(nthcdr 2 '(hylomorphism: all is form & matter))<CR>
;; (nthcdr <index><list>)
;; Starting at 0, return the indexed cdr of the given list.
(form & matter)

Lists can be separated into their components with the functions covered in this section; but, how are they built up?

c. List Construction: **cons**, **append** and **list**

The list selectors **car** and **cdr** separate a list into its "head" or "function" and its "tail" or "arguments". The list constructor **cons** is their dual: it synthesizes a "head" and "tail" into a list. (Winston, 1984, p. 29-31)

<list> ::= **-> (cons** ['<head> ['<tail>**)**<CR>
<list> ::= **(**<head><tail>**)** ::= **-> (cons** '<head> '<tail>**)**<CR>
<head> ::= <LISP form>, and <tail> ::= <list>[35]

Therefore, in order to synthesize a list out of two parts:

-> (cons 'plus '(1 2 3))<CR>
;; (cons '<head> '<tail>)
(plus 1 2 3)

To create lists use **list** with this format:

<list> ::= **-> (list** { ['<LISP form> }***)**<CR>

An example that makes a list out of several arguments is:

-> (list 'This 'is 'a 'joined 'sentence!)<CR>
;; Make a list out of the following elements.
(This is a joined sentence!)

---

[35] In actuality an atom can form the tail element, this produces a dotted list, e.g., **(**<head>.<tail>**)**

In order to "splice" lists together use **append**:

<list> ::= **-> (append** { '<list> | **(list** [']<LISP form>**)**}***)**<CR>

Both **list** and **append** evaluate their arguments, but **append** splices its arguments' values together:

-> (append '(This is) '(not a) '(disjoint sentence.))<CR>
;; Join the lists into one list.
(This is not a disjoint sentence.)

The null list is called **nil**, this is also the LISP word for false[36]:

-> (list '())<CR>
nil

List selectors and constructors break up or join LISP expressions and may be used to rearrange a list's elements. These elements might be LISP functions or their arguments, that manipulated as data, can be placed into a list format in which the function can then be applied to its arguments. This idea is now examined.

d. Functional Application: **apply** and **funcall**

These two functions apply a function to a list or to a set of arguments. The syntax for **apply** is:

<value> ::= **-> (apply** <function-name>
{**(list** {[']<parameter>}***) | '(**<parameter>***)}**)<CR>

apply takes a function and a list of parameters for the function as its arguments, as shown below:

-> (apply 'plus '(1 2 3))<CR>
;; (apply <function-name><parameter-list>)
6

---

[36] See the discussion in Section II.3.e.

-> (apply 'append '((a b)(c d)(e f)))<CR>
(a b c d e f)

funcall is similar to apply, except that it accepts each parameter for the function individually. It has this format:

<value> ::= **-> (funcall** <function-name> { ['']<parameter> }***)**<CR>

Examples of **funcall** now follow:

-> (funcall 'plus 1 2 3)<CR>
;; (funcall <function> { ['']<parameter>}*)
6

-> (funcall 'append '(a b) '(c d) '(e f))<CR>
(a b c d e f)

Up to this point, functions can be applied sequentially to each other; but so far, there is no way to conditionally apply a function. In order to build control structures that can do this, the idea of a predicate is now introduced.

e. Predicates (the Values **t** and **nil**) and the **cond** Control Structure

A predicate is a function whose value is either true or false. The LISP symbol for true is **t** and for false it's **nil**. In LISP any non-**nil** value is considered to be true. Both **t** and **nil** evaluate to themselves. The empty list is also called **nil** and is the only LISP expression that is simultaneously a list and an atom! (Winston, 1984, p. 44-46)

Therefore, the following is true:

{ **t** | **nil** } ::= **->** (<predicate><LISP form>***)**<CR>

Many LISP **p**redicates end with a **p**, e.g. **listp**, **minusp**, etc., but there are important exceptions such as: **atom**, **null** and **equal**. (Touretzky, 1984, pp. 14-17) So, for example:

```
-> (listp '(a b c))<CR>
;; Is "(a b c)" a list?
    t

-> (null 'a)<CR>
;; Is " a " null? I.e., is it equal to nil.
    nil

-> (atom 'a)<CR>
;; Is " a " an atom?
    t

-> (equal nil '())<CR>
;; Is nil equivalent to ().
    t

-> (> 3 2 1 4)
;; Some predicates accept more than one parameter. In this case,
;; " > " checks to see if all the parameters are strictly decreasing.
    nil
```

Predicates in conjunction with the conditional function, **cond**, are used to control execution flow. The conditional is similar to an " IF . . ., THEN . . ., ELSEIF . . ." statement and has the following syntax:

```
(cond ({<test form> <action>}+))
<test form> ::= ( [not] <predicate form>) |
                ( [and | or] {<predicate form> | <test>}*)
<predicate form> ::= (<predicate><LISP form>*)
```

The <test>s are performed sequentially[37] until one evaluates as **t**, then its corresponding action is performed.

The LISP primitive functions **and** & **or** are simple control structures which are used as follows (Foderado, 1983, p. 4-1 & p. 4-13):

---

[37] MacPitts has a conditional form, **cond**, which conducts its tests in parallel, and selects the first one that's true. This mode of operation reflects the VLSI implementation of the conditional function in MacPitts.

-> (and)<CR>
;; If "and" has no arguments it returns t.
t

-> (and 1 2 (plus 2 3))<CR>
;; If all its arguments are non-nil, then "and" gives the value of
;; its last argument; otherwise, if any argument evaluates to nil
;; the result is nil.
5

-> (or)<CR>
;; If "or" has no arguments it returns nil.
nil

-> (or (zerop[38] **1)(* 3 5)())**<CR>
;; Returns the first non-nil value, otherwise if all its
;; arguments evaluate to nil, " or " returns nil.
15

In another example, examine how a predicate, **member?**,[39] is constructed using conditional tests and the LISP function **member**:

-> (member 'a '(b c a d e))<CR>
;; member returns a list that starts with the first instance
;; of the element that is being checked for membership in a
;; list.
(a d e)

The code for the **member?** predicate is now shown. Observe that " list " is a parameter and not the **list** function:

---

[38] **-> (zerop 1)**<CR>
nil
-> (zerop 0)<CR>
t

[39] See Chapter III for a description of lincoln.l. In lincoln.l predicates usually end with a " **?** ".

```
-> (defun member? (element list)
     (cond
        ;; IF the list is null, THEN return nil.
        ;; This is the Basis Condition: it ensures termination.
        ((null list ())
        ;; ELSEIF the element is equal to the head of the list,
        ;; THEN return t
        ((eq (car list) element) t)
        ;; ELSEIF the element isn't equal to the list's head, THEN
        ;; apply this procedure again  to the list's tail.
        ;; The t in this last conditional test means that if the
        ;; other two conditional checks fail, then this last
        ;; statement will always get done.
        ;; This is the Recursive Condition: it ensures all the
        ;; elements get checked for membership.
        ( t (member? element (cdr list))) ))<CR>
        member?

-> (member? 'and '(Self prophecy and recursion))<CR>
;; (member? <element><list>)
;; Is "and" a member of the list "Self prophecy and recursion"?
        t
```

Another example of a function that uses the conditional statement follows. First, the function, **match-that**, is applied in a simple example:

```
-> (match-that 3 '(1 2 3 4 5 6 7) '<)<CR>
;; (match-that <element><list><predicate>)
;; Returns a list composed of elements in the list that satisfy the
;; predicate relation with the thing. Notice that the optional
;; argument was not used here.
   (4 5 6 7)
```

Now the function's code is presented. Again note the use of the parameter " list ". The " tail " variable is were the results are being stored as the recursion unwinds:

```
-> (defun match-that
     (thing list predicate &optional tail)
       (cond
          ;; This is the recursion's basis condition:
          ;; If the list is empty, then all the results are in the tail.
          ;; Since the first elements are being consed into the tail
          ;; first by the application of match-that to the remainder
          ;; of the list, (cdr list), when the basis condition is met,
          ;; all the element in the tail will be backwards.
          ;; Therefore, reverse them and return this as the result.
          ;; This is the Basis Condition: stop if the list is empty.
          ((null? list)(reverse tail))
          ;; The list wasn't empty, therefore, apply the predicate
          ;; to the element's head. If the predicate is satisfied,
          ;; place the head in the list called "tail".
          ;; This is a Recursive Condition: apply the predicate to
          ;; first list element, (car list), and match-that to the
          ;; rest of the list, (cdr list).
          ((funcall predicate thing (car list))
           (match-that
                thing
                (cdr list)
                predicate
                (cons (car list) tail)) )
          ;; Since the list wasn't empty and the head element did'nt
          ;; satisfy the predicate, apply this algorithm to the rest
          ;; of the list. Another Recursive Condition.
          (t
           (match-that thing (cdr list) predicate tail)]40<CR>
;; LISP returns the function's name
        match-that
```

Predicates can also be used in iterative control structures.

---

40 The "] " is a right superparenthesis. A right superparenthesis can substitute for as many regular parenthesis, ") " as would be required to close off the <LISP form>. However, the count stops as soon as a left superparenthesis, " [", is encountered. (Wilensky, 1985, p. 42)

f. Iteration: **prog**, **do**, **do***[41], and **mapcar**

Although recursion is very often used in LISP, there are times when an iterative approach is preferable. LISP has various iterative control structure. One syntax for LISP iteration uses **prog** (Wilensky, 1984, p. 77):

(prog (<local-variable>***)**
{ **(setq** <local-variable><LISP form>**)** | <PROG form> }***)**
<PROG form> ::= <tag> | **(go** <tag>**)** |
(return <LISP form>**)** | <LISP form>
<tag> ::= <atom>

The use of **prog** is like programming in BASIC with its loops and go to's. The way **prog** works is as follows (Winston, 1984, p. 87):

- The first position in a PROG is always occupied by a list of parameters, which are all bound on entering the PROG. Each parameter that has a value before the PROG is evaluated is restored to its previous value upon exit.If there are no parameters, NIL or () must be in the first position. The parameters are each initialized to NIL automatically . . . .

- The forms in the body of a PROG are evaluated one after the other. The values are ignored, so the evaluations are only useful for side effects. If control runs off the end of a PROG, then NIL is returned, just as with COND.

- Whenever a RETURN expression is reached when evaluating a PROG, the PROG is terminated immediately. The value of the terminated PROG is thevalue of the argument in the RETURN expression that stopped the PROG, just as with DO.

- Any top-level symbol in the body of a PROG is considered to be a position marker. These symbols, called *tags* , are not evaluated. They mark places to which control can be transferred by GO expressions. That is, (GO <TAG>) transfers control to the form following the <TAG>.

---

[41] See the comment about **let*** and **let***. **do** and **do*** have the same relation as **let** and **let***. **do** assigns values in parallel, whereas **do*** does it serially.

The **setq**'s are used to assign values to variables within the context of the **prog**. As an example, review this definition of a factorial function:

```
-> (defun factorial (integer)
;; Bind local variables to nil.
    (prog (result)
;; Initialize local variables
      (setq result 1)
;; A loop that will find the factorial of a positive integer.
     loop
;; IF the integer is zero then exit the prog and return the result.
       (cond ((zerop integer)(return result)))
;; OTHERWISE, multiply the integer by the accumulated result,
;; then decrement the integer by one and repeat the loop.
       (setq result (* integer result))
       (setq integer (1-  integer))(go loop) ) )<CR>
   factorial
```

A more structured iterative syntax, which can do everything **prog** does, uses **do** or **do*** (Winston, 1984, p. 86):

(do ({**(**<variable> <initial-value> <update-form>**)**}***)**
(<end-test> <LISP form>* <result-form>**)** <body> **)**[42]
<end-test> ::= <test form>[43]
<result-form> ::= <LISP form>, and <body> ::= <LISP form>*

However, if an action is to be performed across lists, then "the lazy man's do loop", **mapcar**, can be used. (Winston, 1984, p. 79) For example, given the LISP primitive **zerop**, a list's elements can all be checked for equality with zero in one fell swoop:

```
-> (mapcar 'zerop '(1 0 a 0 0 2))<CR>
                  (nil t nil t t nil)
```

---

[42] See Section II.C.4 for an example of **do**.

[43] See Section II.C.3.e for <test form>'s format.

The **mapcar** function applies a function across the first list elements, then across the second list elements, until the shortest list is exhausted. The function must be able to take as many parameters as there are lists. Generalized lambda functions to perform complex operations can be constructed and applied across lists using **mapcar**. Its format is:

(<value>***)** ::= **-> (mapcar** <function-name> { ['']<list>}***)**<CR>

Some examples of **mapcar**:

```
-> (mapcar 'list '(a b c) '(1 2 3) '(X Y Z))<CR>
;; Make a list that has sublists with the respective element in
;; each of the given lists.
    ((a 1 X)(b 2 Y)(c 3 Z))

-> (mapcar '(lambda (x)(plus x 5)) '(1 2 3 4 5))<CR>
;; Add 5 to each list element.
    (6 7 8 9 10)

-> (mapcar
        '(lambda (x y)(times x y))
        '(1 2 3 4 5) '(3 4 5 67) )<CR>
;; Multiply two lists.
    (3 8 15 24 35)
```

LISP's iterative control structures are convenient tools that supplement its naturally recursive style.

4. Iteration and Recursion[44]

In an iterative routine, "indefinite repetition is designated by explicit instructions to do something repeatedly." The **do** construct in LISP is one format for iteration. (Wilensky, 1984, p. 75) An iteratively defined function which raises a number to a power is a good example (Winston, 1984, p. 85):

---

[44] Refer to the discussion in Section II.A.

```
-> (defun m-to-the-n (m n)
     (do[45] ((result 1 (* m result))
              (power n (- power 1)))
             ((zerop power) result)))<CR>
;; Raise a number to a positive power: m^n.
        m-to-the-n

-> (m-to-the-n 2 3)<CR>
    ;; result1 := 1, power := 3, (zerop 3) := nil
    ;; result2 := (* 2 1) := 2, power := (- 3 1) := 2, (zerop 2) := nil
    ;; result3 := (* 2 2) := 4, power := (- 2 1) := 1, (zerop 1) := nil
    ;; result4 := (* 2 4) := 8, power := (- 1 1) := 0, (zerop 0) := t
            8
```

Recursion accomplishes indefinite repetition "by having a function call itself during its execution." (Wilensky, 1984, p. 73) A recursive implementation of **m-to-the-n** (Winston, 1984, p. 64):

```
-> (defun m-to-the-n (m n)
    ;; The exponent [ n ] should be a non-negative integer.
    (cond
      ;; Test to see if the exponent [ n ] is zero,
      ;; if it is, return a value of one.
      ;; This is the Basis Condition.
      ((zerop n) 1)
      ;; if the exponent is not one, then
      ;; multiply m by (m-to-the-n m (1- n)), n.b.,
      ;; the recursion will end since n will be reduced
      ;; to zero and (m-to-the-n m 0) is one!
      ;; This is the Recursive Condition.
      (t (* m (m-to-the-n m (1- n)[46]))) ) ) )<CR>
    m-to-the-n
```

---

[45] Refer to Section II.C.3.e for **do**'s syntax.

[46] **1-** decrements by one, while **1+** increments by one.

```
-> (m-to-the-n 2 3)<CR>
;; First time through: (m-to-the-n 2 3)
;; m := 2, n := 3, (zerop 3) := nil, therefore
;; result1 := (* 2 (m-to-the-n 2 (1- 3))) := (* 2 (m-to-the-n 2 2))
;; Second time through: (m-to-the-n 2 2)
;; m := 2, n := 2, (zerop 2) := nil, therefore
;; result2 := (* 2 (m-to-the-n 2 (1- 2))) := (* 2 (m-to-the-n 2 1))
;; Third time through: (m-to-the-n 2 1)
;; m := 2, n := 1, (zerop 1) := nil, therefore
;; result3 := (* 2 (m-to-the-n 2 (1- 1))) := (* 2 (m-to-the-n 2 0))
;; Fourth time through: (m-to-the-n 2 0)
;; m := 2, n := 0, (zerop 0) := t, therefore
;; result4 := 1, Finally a result! Now, substituting backwards:
;; result3 := (* 2 result4) := (* 2 1) := 2
;; result2 := (* 2 result3) := (* 2 2) := 4
;; result1 := (* 2 result2) := (* 2 4) := 8
        8
```

LISP has a variety of useful control structures and a fecund vocabulary. In addition, LISP has other tools to aid in quickly developing working programs. Some of these are considered in the next section.

D. THE FRANZ LISP PROGRAMMING ENVIRONMENT

LISP has a rich panoply of tools to aid the programmer. Among these are a package for stepping through functions as they are being evaluated; a program for debugging faulty code; and, a facility for tracing functions and the values they are manipulating.

1. Program Development Aids

The stepper, debugger and tracer are normally automatically loaded into LISP when they are needed. They work based on a simple idea: it's sometimes easier to see a mistake as a program is running than to catch a

logical error. The three basic functions associated with these programs are **trace**, **debug** and **step**.[47] To see how they work, recall **factorial**:

```
-> (factorial[48] 5)<CR>
;; 5*4*3*2*1 := 120
      120
```

The operation of **zerop** can be observed using **trace**, as follows:

```
-> (trace zerop)<CR>
[autoload /usr/lib/lisp/trace]
[fasl /usr/lib/lisp/trace.o]
;; The tracer returns a list of functions being traced.
      (zerop)
```

Now, every time that **zerop** is used its associated values are shown:

```
-> (factorial 5)<CR>
1 <Enter> zerop (5)
1 <EXIT> zerop nil
1 <Enter> zerop (4)
1 <EXIT> zerop nil
1 <Enter> zerop (3)
1 <EXIT> zerop nil
1 <Enter> zerop (2)
1 <EXIT> zerop nil
1 <Enter> zerop (1)
1 <EXIT> zerop nil
1 <Enter> zerop (0)
1 <EXIT> zerop t
120
```

---

[47] For discussions of these areas see:
(Foderado, 1983, Chapter 11 [Tracer], Chapter 14 [Stepper], Chapter15 [Debugger] and Chapter 16 [Editor])
(Wilensky, 1984, Chapter 11 [Debugging])
(Charniak, 1985, Section 2.8 [Debugging])
(Winston, 1984, Chapter 14 [Debugging])

[48] Defined in Section II.C.3.f.

If the tracing feature is no longer desired it can be stopped as follows:

```
-> (untrace)<CR>
 ;; The untrace function returns a list of all the functions
 ;; it has stopped tracing.
      (zerop)
```

Sometimes though, an error cannot be localized to any particular function. In that case the **step** function allows the user to observe the incremental operation of a program. The debugger can be entered from the stepper, or vice-versa, or it can be invoked separately. In this case the stepper is invoked so that only interpreted code is shown as follows:

```
-> (step e)<CR>
;; Step only interpreted code. If a "t" argument was
;; provided then all code would be stepped.
[autoload /usr/lib/lisp/step]
[fasl /usr/lib/lisp/step.o]
t

-> (factorial 3)<CR>
;; A <CR> is needed in order to continue stepping. A "q"
;; stops stepping. "p" shows the current form in full. An
;; "n"<integer> steps through the given number of evalua-
;; tions without stopping. A "d" goes into the debugger.
(factorial 3)
  3
(prog (result) (setq result 1) loop (cond (& &))
                ...)
(setq result 1)
     1
   1
   (cond ((zerop integer) (return result)))<CR>
    (zerop integer)
      integer = 3
    nil
   nil
   (setq result (* integer result))c<CR>
   3
```

```
   (setq integer (|1-| integer))n2<CR>
   2
   (go loop)
   (cond ((zerop integer) (return result)))
    (zerop integer)d<CR>
;; Go into debug mode. Usually invoked with: (debug)
[fasl /usr/lib/lisp/fix.o]

<------debug------>
;; Obtain a listing of debug commands using help.
: help<CR>
u / u n / u f / u n f     go up, i.e. more recent
                           (n frames) (of function f)
up / up n                 go up to next (nth) non-
                           system function
d / dn                    go down, i.e. less recent
                           (opposite of u and up)
ok / go                   continue after an error or
                           debug loop
redo / redo f             resume computation  from
                           current frame (or at fn f)
step                      restart in single-step
                           mode
return e                  return from call with
                           value of e (default is nil)
edit                      edit the current stack
                           frame
editf / editf f           edit nearest fn on stack
                           (or edit fn f)
top / bot                 go to top (bottom) of
                           stack
p / pp                    show current stack frame
                           (pretty print)
where                     give current stack posi-
                           tion
help / h / ?              print this table --
                           /usr/lisp/doc/fixit.ref
help ...                  get the help for ...
pop / ^d                  exit one level of debug
                           (reset)
```

bk / bk n / bk f backtrace (to nth frame)
/ bk n f / (of fn f)
..f function names only
..a include system functions
..v show variable bindings
..e show expressions in full
..c go no deeper than here
***** combinations are allowed *****
;; Find out where in the stack the user is at present.
:where<CR>

<------debug------>
you are at top of stack.
there are 1 debug's below.
;; Go down the stack.
:dn<CR>
;; This is the <LISP form> at this stack level:
(eval (debug))
;; Go up to the stack top.
:top<CR>

<------debug------>
:return 5<CR>
;; The user intended to move down the stack and change
;; the value being returned, but instead made a mistake,
;; and therefore interrupted the action:
^C^C^CInterrupt:
Break nil
;; Lisp will now go into an error loop and the stack
;; contents saved up so the user can check them. The
;; showstack function shows the current stack contents.
;; The baktrace function is similar. Within debug
;; baktrace is "bk". "bkfv" in debug would print out
;; function names instead of <LISP forms> and would
;; show variable bindings.
<1>: (showstack)<CR>
break-err-handler
***break**
sys:int-serv
tyi

```
funcall-evalhook*
evalhook*
(zerop integer)
(cond (<**> (return result)))
evalhook
continue-evaluation
funcall-evalhook*
evalhook*
(cond ((zerop integer) (return result)))
(prog (result) (setq result 1) loop ...)
evalhook
continue-evaluation
funcall-evalhook*
evalhook*
(prog (result) (setq result 1) loop ...)
(factorial 3)
evalhook
continue-evaluation
funcall-evalhook*
evalhook*
(factorial 3)
;; The stack has LISP system function calls interspersed
;; with the factorial function. A handy feature of the
;; error loop is that the current variable values can be
;; easily obtained. Showstack returns nil.
nil
;; What is the "integer" variable's value?
<1>: integer<CR>
2
;; What is the "result" variable's value?
<1>: result<CR>
3
;; Leave the error loop.
<1>: (reset)<CR>

[Return to top level]
```

Hopefully, this very brief look at some LISP programming tools will encourage the user to experiment with them. The next section reviews the salient points covered up to now.

2. Summary

Although at first, the fact that LISP functions and data look alike is disconcerting; after a brief period of adjustment, having only one format for everything becomes a strong asset. Additionally, LISP is a mature language that has many program development tools integrated into it.

A lot of other material was also covered in this chapter; however, LISP's key ideas are well stated in this quote from (Brooks, 1985, p. 3).

> • Lisp provides an interactive system in which the user types an expression and Lisp *interprets* it (or *evaluates* it) and prints out the result. Thus, large programs can be built and tested incrementally, and at each stage of testing the full power of Lisp is available to examine the state of the program and data structures. Rather than go through another edit-compile-link-run cycle to test a bug hypothesis, the user can test it directly by typing Lisp statements to the interpreter.
>
> • Lisp programs and data have the same form. An often-touted consequence of this is that Lisp programs can modify themselves. A more important result is that it is very simple to write embedded languages in Lisp. For a particular application, a user often can very quickly write a language (i.e., a translator from the language into Lisp) that is in some way well suited to the problem being solved.
>
> • Lisp systems manage storage allocation for the user by providing a dynamic heap of storage that is allocated for data storage as needed, and then "garbage collected" (i.e. reclaimed) in a manner invisible to the user when no longer needed. The user is freed from worrying *a priori* about how much storage will be needed for a particular procedure over all possible inputs.
>
> • Most Lisp systems include a compiler that compiles programs written in Lisp into efficient machine code. Thus, user programs can be run efficiently. In addition, a user-written embedded language can be compiled into machine code essentially for free; it need only translate user language programs into Lisp.

- Lisp functions (equivalent to subroutines or procedures in other languages) are data objects that can be passed as parameters to other functions. This makes it possible to write extensible control structures in user programs that are very difficult to duplicate in more traditional languages.

III. MACROS, FUNCTIONS AND DATA STRUCTURES: LINCOLN.L

The program lincoln.l includes LISP macros and functions for numeric or string comparison, selection, and manipulation. In addition, lincoln.l contains a data structure macro, **defstruct**, which orders data by fields and creates macros to manage the data.[1]

A. MACROS

Macros are used "to write more readable code". (Wilensky, 1985, p. 180) They provide other advantages listed in this quote from Brooks,1984, p. 195:

- Macros provide a mechanism for writing program-writing programs.

- Macros add an extra layer of interpretation to Lisp. In the interpreter, both layers get [sic] completed, one after the other. In the compiler, one layer gets [sic] done at compile time, and an assembly language program is produced to simulate the second layer at run time.

- Macros provide an efficient mechanism for abstracting the structure of data out of a program.

- Macros provide a mechanism for writing new special forms and control structures.

Macros are used to extend LISP by using data abstraction.

---

[1]The defstruct concept is similar to the "structural primitive" idea which some AI researches naively hoped would lead through a process of generalization to a model of human conceptualization (Dreyfus, 1979, pp. 166-9).

1. Data Abstraction and Macros

Abstraction of low level functions can aid understanding. For example, the unmnemonic **car** might be renamed **head**:

-> (defun head (x)(car x))<CR>
head

-> (head '(A B C D))<CR>
A

The mnemonic quality of this new function is offset by the overhead of having a user defined function calling a LISP system function. The LISP function **car** takes one instruction, but a user defined function takes five or more instructions! (Brooks, 1984, pp. 179-180)

Since data abstraction is an important programming tool, the cost of the extra function calls in compiled code is removed by the use of macros. "A macro is a function which accepts a Lisp expression as input and returns another Lisp expression." (Foderado, 1983, p. 8-3)

A macro is efficient because it creates code that the LISP interpreter evaluates only once. Subsequent calls to the macro use the expanded code (Wilensky, 1984, pp. 180-195). The function **defmacro** [**def**ine **macro**] is one of three ways to create a macro (Foderado, 1983, p. 8-3). For example:

<macro-name> ::=
-> (defmacro <macro-name> **(**<argument>***)**<LISP form>***)**<CR>

<macro-name> ::=
-> (def <macro-name> **(macro (**<argument>**)**<LISP form>***))**<CR>

<macro-name> ::=
->(defun <macro-name> **macro (**<argument>**)**<LISP form>***)**<CR>

A macro is applied just like a function:

<value> ::= **->** **(**<macro-name><parameter>***)**<CR>

These examples show that a macro acts very similary to a function:

```
-> (defmacro head (X) (list 'car X))<CR>
;; Define a macro that finds a list's head.
;; LISP returns the macro's name (recall the use of defun).
        head

-> (head '(A B C D))<CR>
;; A macro is used like a function.
        A
```

The LISP **macroexpand** primitive can be used to look at the code generated by the macro **head**:

```
-> (macroexpand '(head '(A B C D)))<CR>
;; The code that the macro "head" is expanded into is returned:
        (car (quote (A B C D)))
```

Therefore, after the initial macro expansion by the interpreter of **(head X)**, all further calls refer to **(car (quote X))**.

Macros never evaluate their arguments! That's why the macro is written in an awkward form using the **list** primitive: so that when the expression **(list 'car X)** is passed to **eval**, the **X** argument will definitely be evaluated. (Winston, 1984, p. 124)

2. Eval and The Backquote Macro[2,3]

LISP normally operates by applying **eval** to expressions, unless this is inhibited by **quote**. (Winston, 1984, pp. 34-35) The dual effect is created by

---

[2] Refer to the discussion in Sections II.B.1.b & c of **eval** and **quote**.

[3] The backquote character macro is usually associated with " ` ". Because this backwards quote is difficult to distinguish from other diacritical marks, the " **$** " is used in this thesis. This is done with:

```
-> (eval-when (compile load eval)(setsyntax '|$| 'macro
        'back-quote-ch-macro) )
```

the backquote macro: expressions are not evaluated unless specified. (Foderado, 1983, pp. 8-3,8-4) The symbol for inhibiting evaluation is " **$** ", for evaluating " **,** " and for evaluating and splicing into a list " **,@** ". (Wilensky, 1984, p. 202) These symbols can be applied in succession, as composite operators, and are summarized in Table 3.1 below:

TABLE 3.1

BACKQUOTE MACRO SYMBOLS

Symbol	Function
$	Inhibit one level of evaluation
,	Evaluate [within the context of " **$** "]
,@	Evaluate and append
$,** or **,$	No-ops, they can be removed.[4]
,@$( ) or **,@'( )**	No-ops
$(,x)	**(list x)**
$(,x ,@y)	**(cons x y)** [y must be a list]
$(,@x ,@y)	**(append x y)** [x & y must evaluate to lists]
$(,@'x ,@'y)	**(append 'x 'y)** [x and y must be lists]

So for example, if the variable **A** is set to have as its value the list **(1 2 3)**, the effect of " **$** ", "**,**" and "**,@**" can be observed:

-> (setq A '(1 2 3))<CR>
;; The variable "A" is assigned the list " (1 2 3) " as a value.
(1 2 3)

-> $(A ,A ,@A)<CR>
;; " A " is unevaluated, " ,A " is evaluated, " ,@A " is evaluated and
;; spliced into the list structure.
(A (1 2 3) 1 2 3)

---

[4] **-> $((a b) (C D) ,@'(e f) ,@'(G H))**<CR>
;; " ,@' " acts as a composite operator: ,@(quote (<argument>))
;; So, first apply quote, and then " ,@ ".
((a b) (C D) e f G H)

Consider the following expression:

```
-> (append
       (cons (list 'a 'b)  (cons (list 'C 'D) '(e f )))
       '(G H)  )<CR>
   ;; The most deeply nested expressions are evaluated first:
   ;; '(e f) => (e f)
   ;; (list 'C 'D) => (C D)
   ;; (cons '(C D)'(e f)) => ((C D) e f)
   ;; (list 'a 'b) => (a b)
   ;; (cons '(a b) '((C D) e f)) => ((a b)(C D) e f)
   ;; (append '((a b)(C D) e f) '(G H)) => ((a b)(C D) e f G H)
       ((a b) (C D) e f G H)
```

The above LISP expression with **append**, **cons** and **list** is equivalent to:

```
-> $((a b) (C D) e f G H)<CR>
;; The user can use the backquote character macro as a template.
       ((a b) (C D) e f G H)
```

Here is how this result was obtained:

```
(append (cons (list 'a 'b)  (cons (list 'C 'D) '(e f ))) '(G H))
;; substitute for the append:
$(,@(cons (list 'a 'b)(cons (list 'C 'D) '(e f))) ,@'(G H))
;; substitute for the outermost cons:
$(,@$( ,(list 'a 'b) ,@(cons (list 'C 'D) '(e f))) ,@'(G H))
;; substitute for the next cons:
$(,@$( ,$(a b) ,@$( ,(list 'C 'D) ,@'(e f))) ,@'(G H))
;; Eliminate no-ops
$(,@$( ,$(a b) ,@$( ,$(C D) ,@'(e f))) ,@'(G H)
$((a b)(C D),@'(e f),@'(G H))
;; The desired "form":
$((a b)(C D) e f G H)
```

The point of the above exercise is that the final result is much easier to scan than a series of **list**s, **cons**es, and **append**s. The programmer can write code that looks like the desired result at the onset instead of manipulating a series of expressions as was done above.

The backquote macro is frequently used in writing macros. It's used to create a template of the code the macro will provide to **eval**, for example:

```
-> (defmacro head (X) $(car ,X))<CR>
;; Equivalent to: (defmacro head (X)(list 'car X))
        head
```

These ideas are all brought to fruition when functions that generate other functions are made. A good example is the **defstruct** [**def**ine **struct**ure] macro.[5] This macro consists of two levels. The lowest level creates the desired function according to a template. The upper level evaluates the function that was created. A brief sketch and a bit of the LISP code demonstrates the idea:[6]

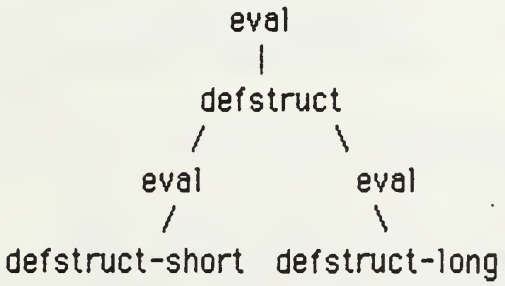

Figure 3.1 The Defstruct Function Hierarchy

The code that follows reflects the structure in Figure 3.1. There is a main **eval-when** form that evaluates the **defstruct** function. This function in turn has two **eval-when** forms in it. They will either evaluate

---

[5] See Section III.C for more detail on **defstruct**.

[6] The reader should skim through this code looking at how the evaluation statements are nested with macro or function definitions. Look at the code's form and the extent that it "shows" the macros it is generating. The LISP function **eval-when** tells the interpreter or compiler to evaluate this code when it is loaded into LISP.

the results of a **defstruct-short** or a **defstruct-long** function. The reader should keep in mind that the results of these functions are in turn other functions. Here is the LISP code:

```
-> (eval-when (compile load eval)
    ;;Evaluate whatever destruct returns:
    (def defstruct (macro (body)
     (cond
       ((listp (caddr body))
       ;; Conditionally evaluate the result of defstruct-short:
         $(eval-when (compile load eval)
             ,@(defstruct-short (cadr body)
                                (caddr body))))
       ;; Conditionally evaluate the result of defstruct-long:
       (t $(eval-when (compile load eval)
             ,@(defstruct-long (cadr body)
                               (cddr body)))))))
    (defun defstruct-short (type fields)
   ;;The defstruct-short function makes "short" structure macros:
    $((def ,(concat 'make- type)
     (macro (x) $$(,,@(cdr x))))
     ,@(defstruct-short-fields type fields 1)
     ,@(defstruct-replace-fields type fields 1)))
    (defun defstruct-long (type body)
    ;;The defstruct-long function makes "long" structure macros:
     (cond
       ((null body)())
       ((or (null (cdr body))(listp (car body)
            (atom? (cadr body)))
        (error '| Invalid defstruct syntax in
                                   defstruct-long |))
       (t
        $(,(make-case-type (car body) type)
          ,(is-type-case? type (car body))
          ,@(defstruct-long-fields type
                        (car body)(cadr body) 2)
          ,@(defstruct-replace-fields
              (concat (car body) '- type)(cadr body) 2)
          ,@(defstruct-long type (cddr body))))))<CR>
```

As an example, a list with fields "name" and "age" will be called a "man". Examining the results from the bottom up shows how functions are first created and then evaluated into the LISP environment. First the lowest level functions **defstruct-short-fields** and **defstruct-replace-fields** create macro definitions in the following fashion:

```
-> (defstruct-short-fields 'man '(name age) 1)<CR>
    ;; Since there are two fields two selector macro definitions
    ;; are made. They are returned in a list.The results are:
    ;; A macro definition that selects the name field: man-name.
    ((def man-name (macro (body)$(car ,(cadr body))))
    ;; A macro definition that selects the age field: man-age.
    (def man-age (macro (body)$(cadr ,(cadr body)))))

-> (defstruct-replace-fields 'man '(name age) 1)<CR>
    ;; Since there are two fields two mutator macro definitions
    ;; are made. They are returned in a list.
    ;; A macro definition that replaces the name field with a new
    ;;  value is created and called replace-man-name.
    ((def replace-man-name (macro (body)$(append
        (list ,(caddr body))(cdr ,(cadr body)))))
    ;; A macro definition that replaces the age field with a new
    ;;  value is created and named replace-man-age.
    (def replace-man-age (macro (body)$(append
        (list (car ,(cadr body)),(caddr body))
        (cddr ,(cadr body)))))  )
```

The above results are now spliced into a list of macros:

```
-> (defstruct-short 'man '(name age))<CR>
    ;; The macro definitions are spliced into a list:
    ((def make-man (macro (body) ... )
     (def man-name (macro (body) ... )
     (def man-age (macro (body)  ... )
     (def replace-man-name (macro (body) ... )
     (def replace-man-age (macro (body) ... ) )
```

The list of macros is evaluated, notice that this is equivalent to using the **defstruct** function as follows:

```
-> (defstruct 'man '(name age))<CR>
;; Only the name of the last macro evaluated is returned:
        replace-man-age
```

The macros can now be used like any other LISP function:

```
-> (replace-man-name '(jim 23) 'mike)<CR>
    ;; Replace the name field of this "man" structure with "mike".
        (mike 23)

-> (man-age '(mike 23))<CR>
    ;; Select the "age" field of this "man" structure.
        23
```

The selector and mutator function interactions with the "man's" "age" and "name" are represented in the following figure. An arrow into the field shows that data is being "deposited" and similarly an arrow from a field represents data that is being "extracted".

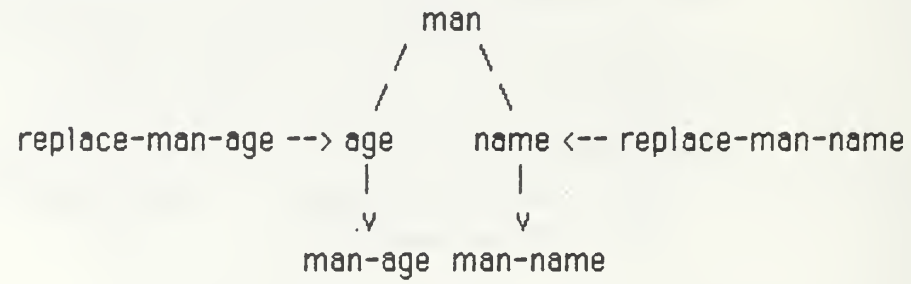

Figure 3.2 "Man" Defstruct Operator Functions

This example, although a bit involved, has shown how LISP macros can be used to create other functions. However, before moving on to the next section, the unconvinced reader should glance at Figure 3.3 and compare it to the definition of **defstruct-long** that used backquote a few pages ago.

There are many handy macros in lincoln.l. These are now examined.

```
(def defstruct-long
 (lambda (type body)
  (cond ((null body) ())
        ((or (null (cdr body))
             (list? (car body))
             (atom? (cadr body)))
         (err '|Invalid defstruct syntax|))
        (t (append (cons (list 'def
                               (concat 'make- (car body) '- type)
                               (list 'macro
                                     '(body)
                                     (list 'cons
                                           "list
                                           (list 'cons
                                                 (list 'list
                                                       "quote
                                                       (list 'quote
                                                             (car body)))
                                                 '(cdr body)))))))
                         (cons (list 'def
                                     (concat 'is- type '- (car body) '?)
                                     (list 'macro
                                           '(body)
                                           (list 'list
                                                 "eq
                                                 (list 'list
                                                       "car
                                                       '(cadr body))
                                                 (list 'list
                                                       "quote
                                                       (list 'quote
                                                             (car body))))))))
                         (append
                          (defstruct-long-fields
                           type (car body) (cadr body) 2)
                          (defstruct-replace-fields
                           (concat (car body) '- type)
                           (cadr body)
```

Figure 3.3 The **defstruct-long** Definition Without Backquote

3. Lincoln.l Macros

Among the wide variety of macros in lincoln.l, one of them, causes inline lambda expressions to be automatically quoted. Recall that **quote** is normally used to inhibit evaluation of LISP forms. **function** is used in a similar fashion to prevent functions from being evaluated. Since most inline lambda expressions require quoting, in lincoln.l **lambda** is defined as a macro that automatically includes **function** in front. (Wilensky, 1984, pp. 119 & 185) Here is its definition:[7]

```
-> (eval-when (compile load eval)
     (def lambda (macro (body)$(function ,body))) )<CR>
```

Several other macros in lincoln.l create mnemonic names for frequently used operations. The **defstruct** macro generates data structures and makes other macros to manipulate the structures. However, the majority of lincoln.l macros are predicates.

a. Numerical Comparison Predicate Macros

The macros **=0**, **<0**, **>0**, **<=0**, **>=0**, **<>0**, **>=**, **<=**, **<>**, and **=1** represent predicates which check the conditions given in Table 3.2. Their syntax, with the exception of **<=** and **=1**, is also used for the MacPitts comparison primitives (Lincoln Lab Report 662, 1983, p. 49). Note that LISP has **<=** and **>=** primitives. LISP primitives for increment (**1+**), decrement (**1-**), add (**+**) , subtract (**-**), equality (**=**), etc., are also used for MacPitts functional syntax. Each of these macros corresponds to a layout primitive

---

[7] **macro** and **nlambda** are also similarly defined.

called an organelle which is covered in Chapter VI. (Siskind, 1982, pp. 14-15)(Lincoln Lab Report 662, 1983, pp. 25-26)

TABLE 3.2

NUMERICAL COMPARISON PREDICATES

Predicate Name	Predicate Test
=0	equality with zero
<0	negative sign
>0	positive sign
>=0	non-negative value
<=0	non-positive value
<=	less than or equal
>=	greater than or equal
<>0	not equal to zero
<>	not equal
=1	equality with one

In addition to the numerical comparison macros shown above, lincoln.l has several macros that perform type checking.

b. Type Predicate Macros

LISP's applicative nature allows functions to be passed as data and provides data handling flexibility at the expense of performing very little type checking.[8] (Gray, P., 1984, p. 111) Whereas in LISP predicates usually have the form <name>**p**, in linconl.l they have the form <name>**?**. Take for example a LISP and a lincon.l predicate that checks if a number is odd:

```
-> (oddp 3)<CR>
;; LISP predicates often end in a " p ".
        t
```

[8] For a discussion of type checking see (Aho, 1986, pp. 343-380).

```
-> (odd? 2)<CR>
;; lincoln.l predicates usually end in a " ? ".
        nil
```

In Franz LISP the function **type** returns one of fourteen types. (Foderado, 1983, pp. 1-1 through 1-6)(Wilensky, 1984, pp. 246-260) Table 3.3 gives a list of lincoln.l predicates and their meanings:

TABLE 3.3

TYPE PREDICATES AND THEIR MEANINGS

Predicate Name	Predicate Meaning
array?	is it an array?
atom?	is it non-nil and not a list?
bignum?	is it an integer greater than a fixnum?
bound?	has it been given a value?
eq?	are they the same structure?
equal?	do they return the same value? (are they equivalent?)
even?	is it even?
fix?	is it a fixnum or a bignum?
fixnum?	is it an integer between -2^{31} and $2^{31}-1$?
flonum?	is it a floating point number?
function?	is it a machine coded function?
list?	is it nil or a list?
null?	is it null?
number?	is it a bignum, fixnum or flonum?
odd?	is it odd?
string?	is it a sequence of characters?
member?	is an element a member of a list?

Macros are used in lincoln.l to create a consistent set of primitives out of many LISP functions that evolved with disparate formats. In addition to homogenizing existing LISP functions with macros, lincoln.l also provides a large number of functions.

B. FUNCTIONS

1. APL Like Operators

APL was one of the first programming languages to apply functions over whole data structures, thereby freeing the programmer from the tedium of iterating over elements.[9] John Backus, one of FORTRAN's creators, wanted to reason algebraically about programs and suggested applying APL's ideas in a purely functional manner. The operations of this algebra would consist of applying, binding, selecting, "composing, reversing, mapping and reducing functions." (MacLennan, 1983, p. 405)

Several functions are shown here as examples of the many useful functions with an APL flavor in this section:

```
-> (such-that '( 0 -1 9 -2 -3) '<0)<CR>
;; (such-that <list> <predicate>)
;; Return all list elements satisfying the predicate.
        (-1 -2 -3)

-> (slash '((a b c)(d c a b)(e f g h)) nil 'union)<CR>
;; (slash <list> <identity> <function>)
;; Return the result of applying a function to a list's elements.
        (a b c d e f g h)

-> (sort '(1 4 2 5 3 9) '>)<CR>
;; (sort <list> <predicate>)
;; Sort a list's elements by a predicate.
        (9 5 4 3 2 1)

-> (car-list '((1 2)(3 4)(5 6)))<CR>
;; (car-list <list>)
;; Find the first element of each of a list's sublists.
        (1 3 5)
```

---

[9] For an excellent APL user's guide see (IBM, 1983, p. 13).

-> (replicate 4 '(man))<CR>
;; (replicate <integer> <LISP form>)
;; Copy a LISP form an integer number of times.
((man)(man)(man)(man))

2. Selection Functions

Franz LISP and APL both have a repertoire of list selectors.[10] Some of lincoln.l's are shown:[11]

-> (nthset 3 '(A B C D) 'e)<CR>
;; (nthset <index> <list> <LISP form>)
;; Replace the indexed position with a given LISP form.
(A B e D)

-> (nthinsert 3 '(a b c d) 'H)<CR>
;; (nthinsert <index> <list> <element>)
;; Insert the element after the indexed position
(a b c H d)

-> (nthdrop 3 '(A B C D))<CR>
;; (nthdrop <index><list>)
(A B D)

-> (nthelem-list
'(1 2 6)
'(Many are called. Few are chosen.))<CR>
;; (nthelem-list <index> <list>)
;; Pick the indexed elements out of a list. Notice that LISP consi-
;; ders a space as the delimiter between atoms [e.g. chosen. or
;; called. are one atom]
(Many are chosen.)

---

[10] See (Foderado, 1983, p. 2-4) and (IBM, 1983, p. 52).

[11] Compare to the selectors presented in Section II.C.3.b.

-> (nthset-list
'(1 4 6)
'(Sad is the woman who cries along the way.)
'(*Happy man sings*))<CR>
;; (nthset-list <index-list> <template-list> <new-element-list>)
;; Replace the indexed positions in the template-list with the
;; respective elements from the new-element-list.
(*Happy* is the *man* who *sings* along the way.)

3. Set Functions

A LISP list can be viewed as a set with elements, e.g.:

{ element1, element2, ..., elementN } :== **(**<element>***)**
<set> ::= <list>

With this point of view in mind lincoln.l provides a variety of set operators:

-> (setify '((a b)(a b c)*(a b)*(a) 2 'a *(a)*))<CR>
;; (setify <list>)
;; Remove redundant elements from a list. Notice that (a b) and
;; (a) occur more than once in the list. Italics are for emphasis.
((a b c)(a b) 2 'a (a))

-> (union '(1 2 3 4) '(2 3 5 4 6 7))<CR>
;; (union $\langle set \rangle_1 \langle set \rangle_2$)
(1 2 3 4 5 6 7)

-> (intersection '(1 2 3 4 5) '(3 4 5 6))<CR>
;; (intersection $\langle set \rangle_1 \langle set \rangle_2$)
(3 4 5)

-> (set- '(1 2 3 4 5) '(2 4))<CR>
;; (set- $\langle set \rangle_1$ $\langle set \rangle_2$)
;; Remove set_2 elements from set_1.
(1 3 5)

4. Numeric Functions

The functions in this section deal mostly with integers or binary numbers:

-> (to-binary 15 7)<CR>
;; (to-binary <integer> <bits>)
;; Create a list that represents the binary equivalent of an
;; integer with the given number of bits.
(0 0 0 1 1 1 1)

-> (to-decimal '(0 0 0 1 0 1 1 0 1))<CR>
;; (to-decimal <binary-number>)
45

-> (ceiling 4.5)<CR>
;; (ceiling <number>)
;; Return the least upper bound integer of a number.
5

-> (floor 4.5)<CR>
;; (floor <number>)
;; Give the greatest lower bound integer of a number.
4

-> (rand 100)<CR>
;; (rand <number>)
;; Return a random integer between zero and the given number.
;; The result generated is a random integer and usually differs
;; with different calls to rand using the same argument.
94

-> (deal 4)<CR>
;; (deal <integer>)
;; Make a randon list of the first four integers.
(4 2 3 1)

-> (deal-list '(1 2 3 4 5 6))<CR>
;; (deal-list <list>)
;; Randomly order a list.
(2 4 1 3 5 6)

The wide spectrum of functions seen in this section crop up throughout MacPitts and LBS. Surprisingly enough though, a large portion of the functions encountered in these programs are generated by one macro: **defstruct**.

C. DEFSTRUCTS[12]

The lincoln.l **defstruct** [**def**ine **struct**ure macro] allows the user to create new data types. It automatically generates macros to create, select, change or type check instances of the data type. The following quote states the idea of a structure (Winston, 1984, p. 100):

> Conceptually, a *structure* is a collection of *fields* and *field values*. We are allowed to define new structures by specifying their particular field names and default field values. We are further allowed to construct individual structures of any already defined type, to access those individual structures, and to revise them. However, in keeping with the spirit of data abstraction, we are not allowed to look at the way individual structuresare represented internally, for we are supposed to be isolated from the actual representation.

Defstructs are frequently used throughout LBS and MacPitts. They are a useful tool when a large number of different data types must be manipulated. The **defstruct** macro creates short or long data structures.

1. Short Defstructs

The short **defstruct** has the following format:

<short form> ::= **(**<field>+ | (<field>*<list> **)**
<field> ::= <symbol>

---

[12] Refer to the examples in Section III.A.2. Lincoln.l's **defstruct** macro is slightly different from those found in other versions of LISP.

Therefore, a short defstruct looks like a list with fields which are all symbols, except for the last field which can be a list. But, there is more to this structure idea than just the list format: evaluating **defstruct** creates a data type with the specified fields. In addition, **defstruct** automatically generates three other macros: one to construct instances of the data type (constructor), another to select field values (selector), and finally one to change field values (mutator). Streching BNF a bit [a short defstruct has three functions and a list of fields], this is represented as:

<short-defstruct> ::= **-> (defstruct** <type> <short form>**)**<CR>
<short-defstruct> ::= <short-selector><short-mutator>
<short-constructor>
(<field value>$^{+}$ | { <field value>*<list> }**)**

a. Short Constructor

To create instances of a data type with parameters for the fields a constructor macro of the following format is used:

<short-constructor> ::= **make-**<type>
(<field value>$^{+}$**)** ::= **-> (make-**<type> ['']<field-value>$^{+}$**)**<CR>

For example:[13]

-> (defstruct point
(name x y layer attributes))<CR>
;; Create a data list of the form: (name x y layer attributes).
;; Defstruct returns the name of one of the macros it creates.
replace-point-attributes

-> (make-point 'in 3 7 'NM '((signal)(river)))<CR>
;; Instantiate a point data type with the given parameters. e.g.
;; name := in, x := 3, y := 7, layer := NM, etc..
(in 3 7 NM ((signal)(river)))

---

[13] A **point** is fully described in Chapter IV.A.2.

Notice that the result is a list with all the field values placed in the order they were entered.

b. Short Selector

Defstruct also creates selector macros to obtain field values. A short selector macro that picks out <field>$_i$ of a <type> short defstruct has the format:

<short-selector> ::= <type>-<field>$_i$
<field-value>$_i$ ::=
-> **(**<type>-<field>$_i$ {'**(**<field-value>$^+$**)** |
(list [']<field-value>$^+$**)**} **)**<CR>

For example:
-> (point-name '(in 3 7 NM ((signal)(river))))<CR>
;; Get the point's name:
in

-> (point-attributes '(vss -2 7 ND ((power)(out))))<CR>
;; Get the point's attributes:
((power)(out))

c. Short Mutator

The third macro automatically generated for a short **defstruct** is used to change field values. Mutators replace a <type> **defstruct**'s <field-value>$_i$ with <field-*new*-value>$_i$ and have the following form:

<short-mutator> ::= **replace**-<type>-<field>$_i$
(<field-value>$_1$...<field-*new*-value>$_i$ ...<field-value>$_N$**)** ::=
-> (replace-<type>-<field>$_i$
{ '**(**<field-value>$^+$**)** | **(list** [']<field-value>$^+$**)** }
{ [']<field-*new*-value>$_i$ }**)**<CR>

The following examples illustrate the use of mutators:

```
-> (replace-point-layer
        '(in 3 7 NM ((signal)(input)))
        'NP)<CR>
        ;; Replace the point's layer with NP:
        (in 3 7 NP ((signal)(input)))

-> (replace-point-y
        '(in 3 7 NM ((signal)(river-router)))
        11)<CR>
        ;; Replace the point's y coordinate with 11:
        (in 3 11 NM ((signal)(river-router)))
```

Short **defstruct**s are an application of the principle of data abstraction to a list. Each field is given a name, and functions which use those names to manipulate the structure are automatically created. This idea is carried one step further in the long **defstruct** to allow differentiating between closely related structures.

2. Long **defstruct**s

An extension of the short structure concept which allows the fields to be data structures and includes type checking is the long structure. The type is the *genus*[14] and the cases the *species*. The long structure's syntax is superficially the same as a short structure format:

```
<long form> ::= { <case> (({<case-field>+ | <case-field>*<list>}) }+
<case-field> ::= <symbol>
<long-defstruct> ::= -> (defstruct <type><long form>)<CR>
<long-defstruct> ::= <long-constructor><long-selector>
                       <long-mutator><long-interrogator>
                     {(<case> {<case value>+ | <case value>*<list>})}+
```

---

[14] "In Aristotelian logic, a very wide and comprehensive class or kind, subclasses of which may be called species." (Flew, 1979, p. 131)

In both the short and long structure cases **defstruct** is used. A long **defstruct** example with a **tree** genus and eight species:

```
-> (defstruct tree
        null ()
        rect (layer left bottom right top)
        symbol-call (name)
        move (tree dx dy)
        rotcw (tree)
        rotccw (tree)
        mirrorx (tree)
        mirrory (tree))<CR>
```

This long structure creates a **tree** data type. There are eight **tree** cases: **null**, **rect**, **symbol-call**, **move**, **rotcw**, **rotccw**, **mirrorx** and **mirrory**[15]. Note that five of the **tree** cases have a **tree** in their field. The field arguments are also **defstruct**s! A long structure has four associated functions: constructors, selectors, mutators and interrogators.

a. Long Constructor

As in the short structure, macros to construct data type instances are automatically generated in the long structure. A constructor that instantiates the species <case$_i$>-<type> has this format:

<long-constructor> ::= **make**-<case>$_i$-<type>
(<case>$_i$ <<case>$_i$-field-value>***)** ::=
    -> (make-<case>$_i$-<type> { [']<<case>$_i$-field-value>}$^+$**)**<CR>

---

[15]These eight cases correspond to eight basic operations on rectangles. **Null** is no action or no rectangle. **Rect** is a rectangle with the given layer and dimensions. **Symbol-call** represents a method for generating hierarchical representation. **Move**, **rotcw**, **rotccw**, **mirrorx** and **mirrory** represent respectively a displacement by dx and dy; ninety degree clockwise and counterclockwise rotation; and a flip about the x axis or the y axis. The operators these trees represent are described in (Crouch, 1984, p. 8).

This is best shown in a few examples:

```
-> (make-rect-tree 'ND 0 1 2 3)<CR>
       (rect ND 0 1 2 3)

-> (make-null-tree)<CR>
       (null)

-> (make-move-tree '(rect ND 0 1 2 3) 5 9)<CR>
       (move (rect ND 0 1 2 3) 5 9)
```

b. Long Selector

In order to pick <field>$_i$'s value out of a <case>$_i$-<type> species the following format is used:

```
<long-selector> ::= <case>i-<type>-<field>j
<case>i-<field>j-value ::=
   -> (<case>i-<type>-<field>j
         {'(<case>i <<case>i-field-value>+) |
          (list { [']<case>i }{ [']<<case>i-field-value> }+ ) } )<CR>
```

A few examples can make this clearer:

```
-> (rect-tree-layer '(rect NI 1 2 3 4))<CR>
       NI

-> (move-tree-tree '(move (rect NI 1 2 3 4) 5 9))<CR>
       (rect NI 1 2 3 4)
```

c. Long Mutator

Field values are modified in a fashion similar to the short structure mutator, in order to replace <field>$_j$ of species <case$_i$>-<type> a long mutator must be used as follows:

```
<long-mutator> ::= replace-<case>i-<type>-<field>j
(<case>i
    <<case>i-<field>1-value> ...
    <<case>i-<field>j-new-value> ...
    <<case>i-<field>N-value> ) ::=
       -> (replace-<case>i-<type>-<field>j
             {'(<case>i <<case>i-field-value>+) |
              (list { [']<case>i }{ [']<<case>i-field-value> }+ ) }
             {[']<<case>i-<field>j-new-value>} )<CR>
```

This is best seen in a few examples:

```
-> (replace-rect-tree-top '(rect NM 1 2 3 4) 15)<CR>
;; replace a "rect-tree" species' "top" field with 15.
       (rect NM 1 2 3 15)

-> (replace-move-tree-dy
       '(move (rect 1 2 3 4) 9 8)
       11)<CR>
;; replace a "move-tree" species' "dy" field with 11.
       (move (rect 1 2 3 4) 9 11)
```

The **tree** example has shown that a long structure adds a level of complexity to the **defstruct** concept. Why bother? Because there is a big advantage to be gained in grouping similar ideas together and then differentiating between them. In order to do this a long **defstruct** also creates interrogators.

d. Long Interrogator

Long structures offer a limited form of data type checking with their interrogator macros. A check to see if a structure is a <case>i-<type> species can be made as follows:

```
<long-interrogator> ::= is-<type>-<case>i?
{t | nil} ::=
      -> (is-<type>-<case>i?
            {'(<case>x <<case>x-field-value>+) |
              (list { ['']<case>x }{ ['']<<case>x-field-value> }+ ) })<CR>
```

For example:

```
-> (is-tree-rect? '(rect ND 1 2 3 4))<CR>
    ;; is " (rect ND 1 2 3 4) " a "rect-tree" ?
        t

-> (is-tree-null? '(rect ND 1 2 3 4))<CR>
    ;; Is " (rect ND 1 2 3 4) " a "null-tree" ?
        nil

-> (is-tree-move? (move (rect ND 1 2 3 4 ) 5 9))<CR>
    ;; Is " (move (rect ND 1 2 3 4) 5 9) " a "move-tree" ?
        t
```

Keep in mind that the interrogator macro only checks that the correct case name is at the head of the list which composes the data structure. In other words, no check is being made to ensure the correct value types are being placed in the field slots, or even that the structure has the correct number of fields!

3. General Field Structure Checks

defstruct checks its first two arguments to determine whether a short or long format is required. If its first argument is an atom and the second argument a list then a short format is made. If its first argument is an atom and the second argument is also an atom then a long format is made. Otherwise, an error message is printed if the fields are null or the first argument is a list.

The other check that is made ensures that only the last field in a **defstruct** is a list. This occurs in **defstruct-short-fields** and **defstruct-long-fields** where the fields are also checked to be not empty.

4. Summary

defstruct offers the programmer a tool for data abstraction. This idea along with the mnemonic character of constructors, selectors, mutators and interrogators are great aids in data manipulation. **defstruct**s are extensively used in LBS and MacPitts. It might also be speculated that to some degree the mind-body paradigm is reflected in MacPitts' function-data language and controller--data-path architecture. In any case, Table 3.4 presents a **defstruct** summary:

TABLE 3.4

DEFSTRUCT FUNCTION SUMMARY

Function	Short Structure	Long Structure
Constructor	**make**-<type>	**make**-<case>$_i$-<type>
Selector	<type>-<field>$_i$	<case>$_i$-<type>-<field>$_j$
Mutator	**replace**-<type>-<field>$_i$	**replace**-<case>$_i$-<type>-<field>$_j$
Interrogator	None	**is**-<type>-<case>$_i$**?**

IV. LINCOLN LABORATORY LISP LAYOUT LANGUAGE: L5

L5 is a LISP based language for hierarchical representation and manipulation of VLSI circuits. It uses the basic predicates, functions and data structures provided in lincoln.l to create operators for manipulating rectangles and points. With these two building blocks more complex structures can be built. These constructed units can then be used as basic blocks to create other structures in a hierarchical fashion.

A. GLOBAL VARIABLES AND DATA TYPES

Global variables in L5 are used to determine aspects of the technology the circuit will be implemented in; and, thereby constrain the permissible ways data is manipulated. There are several data types in L5. They are created and handled in a consistent manner using **defstruct**[1].

1. Global Variables

There are several global variables in L5 that toggle other processes and thereby affect the behaviour of LBS or MacPitts. The most important ones deal with setting technology dependent factors such as mask layers, process dimensions [Microns/lambda: μ/λ], etc.. The user is provided with functions to access these global variables and check or modify their status. Table 4.1 lists global variables and their functions as a convenient reference:

---

[1] Refer to Chapter III.

TABLE 4.1

GLOBAL VARIABLES AND THEIR FUNCTIONS

Variable	Status Check	Status Modifier & Options
--L5-symbol-storage	**(L5-symbol-storage)**	**(L5-symbol-storage!** ['](<**on-disk** | **in-memory**>)**)**
--L5-technology	**(technology)**	**(technology!** ['](<**nmos** | **cmos** | **cmos-pw** | **cmos3** | **sos**| **scmos**>)**)**
--L5-minimum-feature-size	**(minimum-feature-size)**	**(minimum-feature-size!** <centi-μ per λ>**)**
--L5-symbol-list	**(L5-symbol-list)** & **(create-called-symbol-item** <position>**)**	**(add-symbol-to-L5-symbol-list** <symbol>**)**
--L5-symbol-number	**--L5-symbol-number**	**(setq --L5-symbol-number** <integer>**)** & **(symbol-number)**
--L5-symbol-port	**(L5-symbol-port)**	**(setq --L5-symbol-port** <port>**)**
--L5-symbol-file	**(L5-symbol-file)**	**(setq --L5-symbol-file** <file>**)**
**	**(allowed-layers)**	**
**	**(allowed-conducting-layers)**	**
**	**(layer-table)**	**
%%	**(allowed-technologies)**	**

All of the global variables can be changed using **setq**. Functions with, **, operate by checking the technology global variable and returning an appropriate response without setting any variables. The, **, denotes that to change the values returned by these functions the LISP source code has to be

modified. Finally, the operator with, **%%**, is a constant function that doesn't set any variables and returns.

(nmos cmos cmos-pw cmos3 sos scmos)

Global variables may have different default values:

TABLE 4.2

GLOBAL VARIABLE DEFAULT VALUES

Global Variable	Default Value
--L5-symbol-storage	**on-disk**
--L5-symbol-port	**nil**
--L5-symbol-file	**nil**
--L5-symbol-list	**nil**
--L5-symbol-number	**nil**
--L5-technology	**nmos**
--L5-minimum-feature-size	**250**
--L5-read-stack	**nil**

When the function **L5-symbol-file** [or **L5-symbol-port**] checks its associated global variable's value and finds it to be **nil**, then the symbol file [or symbol port] is changed so that it's located in the **/tmp** directory. The file name is formed by concatenating the current UNIX® process number with an acronym for L5 symbol, " L5sym ". Therefore, global variables operate in the following fashion:[2]

% ps<CR>
# Give the current UNIX® process number.
12904

% lisp<CR>
[Franz Lisp Opus 38.69]

---

[2] A discussion of the symbol list is postponed until the **symbol defstruct** in this section and the **defsymbol** function in Section IV C.1.

-> (L5-symbol-file)<CR>
;; This file is used to output CIF[3] results.
/tmp/L5sym12904

-> (L5-symbol-port)<CR>
;; A port is a LISP I/O device.
%/tmp/L5sym12904

-> (technology! 'cmos)<CR>
;; Set the technology to cmos and list out its layers.
;;These symbols correspond to CIF layers, e.g. CD = n-type
;;diffusion, CP = polysilicon, CM = first layer metal, etc..
(CD CP CM CM2 CS CC CG CW NX XP)

-> (technology)<CR>
;; The current technology is complementary metal oxide
;; semiconductor
cmos

-> (technology! 'scmos)<CR>
;;These are Calma scalable cmos CIF layers, e.g. CMS =
;;metal2, CMF = metal1, CPG = polysilicon, etc..
(CMS CMF CPG CAA CVA CCP CCA CWP CWN
CSP CSN CO6)

-> (minimum-feature-size! 50)<CR>
;; Set 50 centimicrons to be 1 lambda unit.
50

-> (minimum-feature-size)<CR>
;; Currently 50 centimicrons are 1 lambda unit.
50

---

[3] "The Caltech Intermediate Form (CIF Version 2.0) is a means of describing graphic items (mask features) of interest to LSI circuit and system designers." (Mead, 1980, p. 115) Also see (Sequin, 1980, Chapter 7) and (Scott, 1986, Magic Tutorial #9 and Magic Technology Manual #1-2).

```
-> (allowed-technologies)<CR>
;; These are the technologies for which CIF layers have
;; been entered into L5.
        (nmos cmos cmos-pw cmos3 sos scmos)

-> (allowed-layers)<CR>
;; Return the current technology's layers:
        (CMS CMF CAA CVA CCP CCA CWP CWN
                                    CSP CSN C06)

-> (pp allowed-conducting-layers)<CR>
;; Pretty print the definition of the function " allowed-
;; conducting-layers ". Examining the result reveals that
;; this function is simply a large conditional statement
;; that checks the current technology and returns a list of
;; conducting layers. To add another set of conduction
;; layers, simply add the new technology name to the list
;; in the body of allowed-technologies and add a
;; statement in allowed-conducting-layers of the form:
;; ((eq '<new-tech>) (technology)) '(<layer-list>))
(def allowed-conducting-layers
  (lambda ()
    (cond
      ((eq 'nmos (technology))'(NM NP ND))
      ((eq 'cmos (technology))'(CM CP CD CM2))
      ((eq 'cmos3 (technology))'(CM CP CD CM2))
      ((eq 'sos (technology))'(SM SP SIS))
      ((eq 'cmos-pw (technology))'(CM CP CD))
      ((eq 'scmos (technology))'(CMS CMF CP6 CSP
            CSN))
      (t (L5-err '| That technology is not
           recognized by L5|)))))
```

Global variables and their effects will again be encountered in Section IV.C, in the meantime, a look is taken at L5's data types.

2. L5 Data Structures

All L5 data structures are created using **defstruct**. The generic object in L5 is called an **item** and is composed of rectangles and labels. (Crouch, 1983, p. 2) Since an **item** is a grouping of smaller objects it is surrounded with an imaginary rectangle [box] which encompasses all its elements. The smallest box which encloses an **item** is called the Minimum Bounding Box [MBB]. (Ayres, 1983, p. 64) The syntax for an **item** is:

TABLE 4.3

AN ITEM'S SYNTAX

Category	Syntax
<item> ::=	**(**<left><bottom><right><top><points><called-symbol-names><tree>**)**
{<left>|<bottom>|<right>|<top>} ::=	<number>
<points> ::=	**(**<point>*__)__**
<point> ::=	**(**<name><x><y><attributes>**)**
<attributes> ::=	{ **(**<symbol>*__)__** | **(**{**(**<symbol>**)**}*__)__** }
<called-symbol-names> ::=	**(**<number>*__)__**
<tree> ::=	{ <null-tree> | <rect-tree> | <symbol-call-tree> | <move-tree> | <rotcw-tree> | <rotccw-tree> | <mirrorx-tree> | <mirrory-tree> }
<null-tree> ::=	**(null)**
<rect-tree> ::=	**(rect** <layer><left><bottom><right><top>**)**
<symbol-call-tree> ::=	**(symbol-call** <number>*__)__**
<move-tree> ::=	**(move** <tree>**)**
<rotcw-tree> ::=	**(rotcw** <tree>**)**
<rotccw-tree> ::=	**(rotccw** <tree>**)**
<mirrorx-tree> ::=	**(mirrorx** <tree>**)**
<mirrory-tree> ::=	**(mirrory** <tree>**)**

An **item** structure contains two other structures within it: a list of **point** short structures and a **tree** long structure. First, a look at the **item** structure and the creation of a simple **item**:

```
-> (defstruct item
            (left bottom right top
              points
              called-symbol-names
              tree))
        ;; The left, bottom, right and top fields describe the MBB,
        ;; i.e., xmin, ymin, xmax, ymax. The points field contains a
        ;; list of points (labels). The next field is a list of digits
        ;; indicating which symbols on the LS-symbol-list are
        ;; used by this item. The tree field is a list summarizing
        ;; operations performed on the item.
            replace-item-tree
```

Since an **item** is a short **defstruct** it comes with constructor, selector and mutator macros. For example, to create an **item** composed of a metal and a diffusion rectangle with two point labels:

```
-> (make-item 1 2 3 4
      '(((in) 1 2 NM (power))
        ((out) 2 2 ND (external)))
      nil
      ((rect NM 1 2 2 3)
       (move (rect ND 0 0 1 1) 2 3))) )
;; Make an item with a MBB with coordinates (1 2) and
;; (3 4), an "in" label at (1 2) on metal, an "out" label
;; at (2 2) on diffusion, and no symbol calls. The
;; primitives this item is composed of are:
;; (1) A metal rectangle with coodinates (1 2) and (2 3)
;; (2) A diffusion rectangle with coordinates (0 0) and
;;     (1 1) that has been translated to the right 2 units
;;     and to the top 3 units.
      (1 2 3 4
            '(((in) 1 2 NM (power))
              ((out) 2 2 ND (external)))
            nil
            ((rect NM 1 2 2 3)
             (move (rect ND 0 0 1 1) 2 3)) ) )
```

As is seen in the above example, a list of points is a field in an item. The **point** short structure is implemented as follows:

```
-> (defstruct point (name x y layer attributes))
;; A point is a label. Points have names, are located at
;; a specific x and y location and are attached to a layer.
;; A point's attributes can give descriptive information
;; to guide functional application. For example: points
;; with the attribute "external" are actually plotted when
;; CIF is created; the power attribute is used by the
;; function power-line-positions [in the MacPitts program
;; organelles.l] to find Vdd or Vss locations . These
;; positions are then used by layout-metal-lines [in
;; organelles.l] to lay down a metal line grid.
        replace-point-attributes
```

An example now shows the creation of a **point**:

```
-> (make-point '(in) 1 2 'CM
                          '((power)(external)))
;; Make a point whose name is "in", located on CMOS metal
;; at (1 2), and with "power" and "external" attributes.
        ((in) 1 2 CM ((power)(external)))
```

An item's fifth field is a summary of other items used to construct the item. This <called-symbol-names> field is composed of a list of numbers. These numbers represent **symbol**s. A **symbol** is a structure containing an item's salient information. Computer time and memory use are reduced when frequently used items are constructed once and then referred to whenvever needed. Whenever an item is made using the **defsymbol** function [See Section IV.C.1], a pseudo-item, a **symbol**, is placed in the **L5-symbol-list**. Any use of this item will be reflected in the <called-symbol-names> field; these numbers indicate a **symbol**'s position in the **L5-symbol-list**. A **symbol** has the following structure:

```
-> (defstruct symbol
     (ID left bottom right top points
       internal-symbols nest-level tree))
;; Symbols are used by the defsymbol[4] function. Defsym-
;; bols are items that are immediately stored in CIF
;; format in the L5-symbol-file.[5] A symbol representing
;; the item is then placed on the L5-symbol-list. The L5-
;; symbol-list has a symbol for each defsymbol that
;; has been called; consequently, future calls to a
;; defsymbol with the same parameters are referred to in
;; the calling item's called-symbol-names position [a list
;; of numbers giving the position in the L5-symbol-list of
;; the symbol representing the called defsymbol].
     replace-symbol-tree
```

An **item**'s final field is a **tree**, with the following structure:

```
-> (defstruct tree
     null ()
     rect (layer left bottom right top)
     symbol-call (name)
     move (tree)
     rotcw (tree)
     rotccw (tree)
     mirrorx (tree)
     mirrory (tree))
;; A tree is a representation for an item operator[6]. An
;; item's tree is a summary of all the operations
;; performed on the item. A macro name is returned.
     replace-mirrory-tree-tree
```

L5's major data structures are **item**s, **point**s, **symbol**s and **tree**s. They provide a framework for manipulating geometric objects.

---

[4] See Section IV.C.1 for more on **defsymbol.**

[5] This occurs only when the L5-symbol-storage is set to "on-disk".

[6] See Section IV.B.2 for a discussion of item operators.

B. ITEMS AND THEIR OPERATIONS

The **item** data structure is the basic building block in L5. However, having to use the **make-item** function can be a bit tedious. Therefore, L5 has primitive functions for creating rectangles [or boxes] and marks [a **point** that has an **item** format]. L5 also has operators for moving, rotating, etc., **item**s and their **point**s. **Item**s and **mark**s can be grouped together to form larger units using the **merge** function.

1. Item Creation

L5 has four functions for creating primitive items: **null-item**, **rect**, **box** and **mark**:

TABLE 4.4

FOUR PRIMITIVE ITEM CREATING FUNCTIONS

Function	Arguments
null-item	none
rect	<layer><x_{min}><y_{min}><x_{max}><y_{max}>
box	<layer><length><width><x_{center}><y_{center}>
mark	<name><x><y><layer><attributes>

Some examples of these primitive functions are:

-> (null-item)<CR>
;; A null item is useful as a default value for a conditional since
;; it has an item's format with only null fields [Crouch, 1983, p.5]
(nil nil nil nil nil nil (null))

-> (rect 'CD 0 1 4 8)<CR>
;; A rectangle has no points or symbol calls. It consists of its
;; MBB coordinates (0 1) and (4 8) and a rect-tree.
(0 1 4 8 nil nil (rect CD 0 1 4 8)[7]**)**

[7] Note the difference between the <LISP form>, **(rect 'CD 0 1 4 8)**, and the <expression>, **(rect CD 0 1 4 8)**. The first is a function, the second is a data object. The first evaluates its arguments, the second is a list of parameters. Refer to Section II.C.1.

-> (box 'CM 2 8 6 4)<CR>
;; A box is an alternate method of defining rectangles. It's
;; similar to the way CIF defines boxes.
(5.0 0.0 7.0 8.0 nil nil (rect CM 5.0 0.0 7.0 8.0))

-> (mark 'in 5 6 'CMF '(external))<CR>
;; A mark is a point that has an item shell built around it. Notice
;; that the name is automatically converted to a list whereas in
;; using make-point it had to be input as a list[8]. The name can be
;; an atom or a list of atoms. Attributes can be either a list of
;; atoms or a list of lists: (<atom>*) or (<list>*).
(5 6 5 6 (((in) 5 6 CMF (external))) nil (null))

With rectangles and labels any Manhattan geometry can be represented by joining or moving these basic elements.

2. Item Operators

L5's flexibility is due to the many functions that it contains for performing common layout operations so that complex structures can be built up hierarchically from simple building blocks. [Crouch, 1983, pp. 8-11][9] Many of these operations either move or join objects.

a. Translation and Merging Operators

All layouts in L5 are referenced to an imaginary grid in lambda units with center at (0 0). The next group of functions work within this Cartesian framework to assemble or shift items:

---

[8] L5 functions which search among points in an item assume that a point's name is a list. See the functions **find** or **align** in Section IV.B.2.

[9] For a discussion of desirable operators see [Ayres, 1983, pp. 84-88].

TABLE 4.5

TRANSLATION AND MERGING OPERATORS

Function	Arguments	Description
move	<item><dx><dy>	move an item by dx and dy units
home	<item>	place item's top left at (0 0)
first-quadrant	<item>	place item's bottom left at (0 0)
second-quadrant	<item>	place item's right bottom at (0 0)
third-quadrant	<item>	place item's right top at (0 0)
fourth-quadrant	<item>	same as home
merge	<item>+	make one item out of several items
merge-list	(<item>+)	make one item out of a list of items
align	<item> <point-name> <coordinate>	move an item so that the named point is placed on the given coordinate
align-items	$<item>_1$ $<point\text{-}name>_1$ $<item>_2$ $<point\text{-}name>_2$	$<item>_2$ is moved so that its named point aligns with $<item>_1$'s point
rotcw	<item>	rotate 90° clockwise about (0 0)
rotccw	<item>	rotate 90° counter-clockwise . . .
mirrorx	<item>	mirror about the x axis
mirrory	<item>	mirror about the y axis

A brief look at the application of some these functions follows:

```
-> (move
        '(0 0 10 10 nil nil (rect NM 0 0 10 10)) 3 5)<CR>
;; Move the metal rectangle to the right 3 units and up 5 units.
;; Notice how only the MBB is changed [addition and consing
;; elements into a list are fast]. I.e. The result of the operation
;; could have been: (3 5 13 15 nil nil (rect NM 3 5 13 15)), but
;; if the tree was composed of many elements then each one
;; would need to be moved also!
    (3 5 13 15 nil nil (move (rect NM 0 0 10 10) 3 5)
```

```
-> (let
    ;; Set test-item := (-3 0 6 12 (((vss) ...) nil ((rect...)(rect...)))
    ((test-item
     '(-3 0 6 12
        (((vss) 1 2 NM (external))((in) 3 4 ND nil))
        nil
        ((rect NM -3 0 0 4)(rect ND 0 0 6 12)) ) ) )
    ;; Move test-item so the vss point is at (0 0).
    (align test-item 'vss '(0 0)) )<CR>
;; The result:
    (-4 -2 5 10
        (((vss) 0 0 NM (external))((in) 2 2 ND nil))
        nil
        (move
              ((rect NM -3 0 0 4)(rect ND 0 0 6 12))
               -1 -2))
```

Before proceeding with other examples a primitive layout item available in MacPitts, a **layout-inverter**, [Figure 4.1] is introduced:

```
-> (layout-inverter 4 t)[10]<CR>
;; This is an inverter defsymbol composed of several cuts which
;; are also defined as defsymbols. Note that the item's tree is
;; a symbol call, indicating that this item has a symbol in the L5-
;; symbol-list. The item is composed of <symbol>s1,4,6,7 and is
;; itself <symbol>7.
    (0 -20 20 0
        (((gnd) 18 -18 NM (power))
         ((in1)  14 -20 NP  (in))
         ((vdd)   8   -2 NM (power)))
        (1 4 6 7)
        (symbol-call 7))
```

---

[10] The **layout-inverter** function is found in organelles.l [part of MacPitts]. Its syntax is: **(layout-inverter** <pullup/pull-down ratio><mark>**)**

<mark> ::= { **t** | **nil** }

Toggling <mark> to **t** places a label at the "in" point:

```
(mark 'in1 14 -20 'NP '(in)))
```

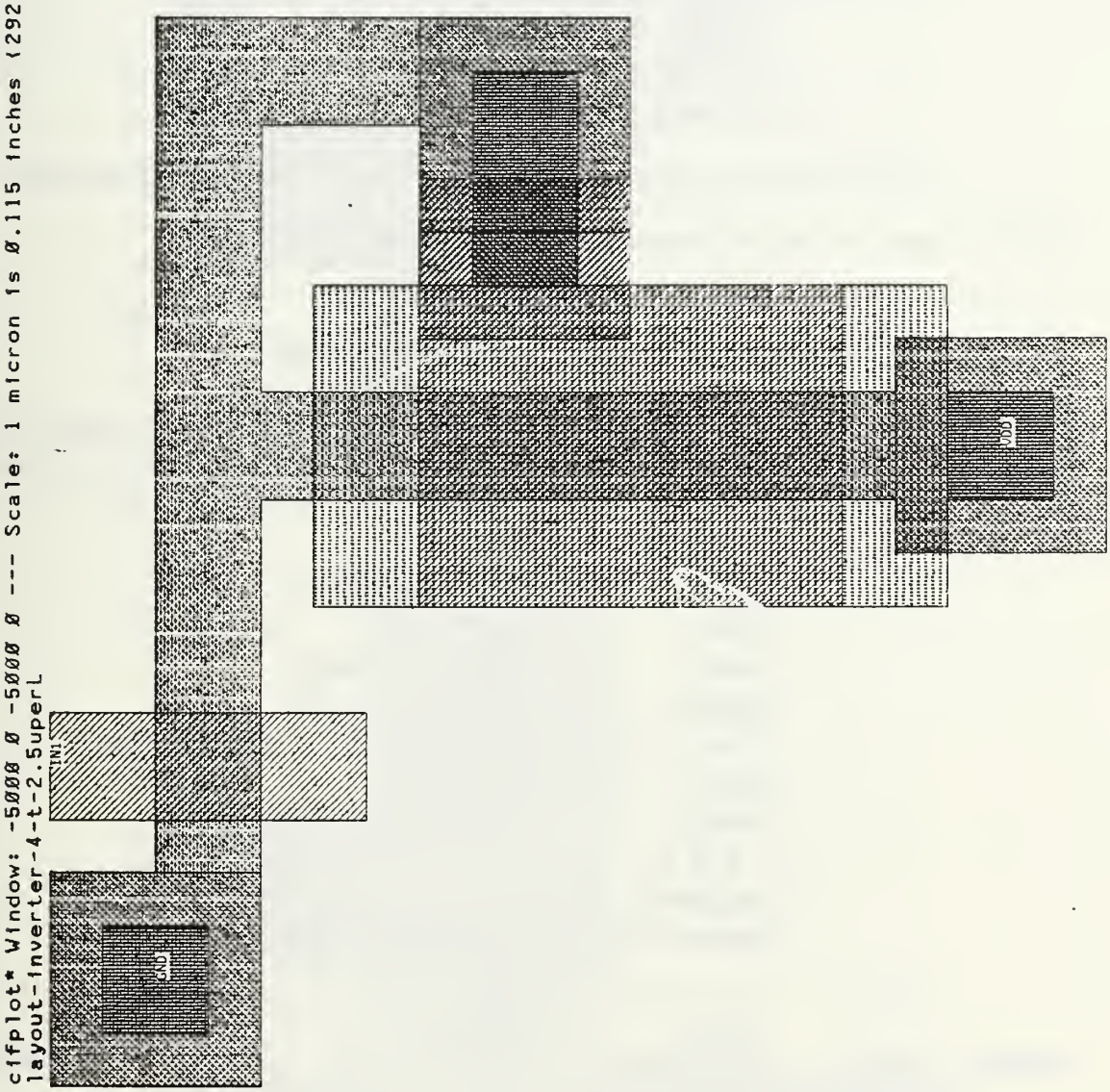

Figure 4.1 **(layout-inverter 4 t)**

This item can be manipulated as follows:

```
-> (mirrorx (layout-inverter 4 t))<CR>
;; Flip the inverter about the x axis.
    (0 0 20 20
        (((gnd) 18 18 NM (power))
         ((in1)  14 20 NP  (in))
         ((vdd)   8  2 NM (power)))
        (1 4 6 7)
        (mirrorx (symbol-call 7)) )
```

More complex objects can be created out of previously defined items as seen in the next examples [See Figures 4.2 and 4.3]:

```
-> (let
    ((inverter (layout-inverter 4 t)))
    (align-items
            inverter 'vdd (mirrorx inverter) 'vdd))<CR>
;; Align the inverter and its mirror image so they have a common
;; Vdd power point.
    (0 -20 20 16
        (((gnd)  18 -18 NM (power))
         ((in1)  14 -20 NP  (in))
         ((vdd)   8  -2 NM (power))
         ((gnd)  18  14 NM (power))
        ;; The next point was cutoff by the plotting routine.
         ((in1)  14  16 NP  (in))
         ((vdd)   8  -2 NM (power)) )
        (1 4 6 7)
        ((symbol-call 7)
         (move (mirrorx (symbol-call 7)) 0 -4)))
```

The next item, shown in Figure 4.3, illustrates the use of the **merge** operator. Notice that this item has the output of one inverter connected to the Vss power source; and as such, is a non-operational layout.

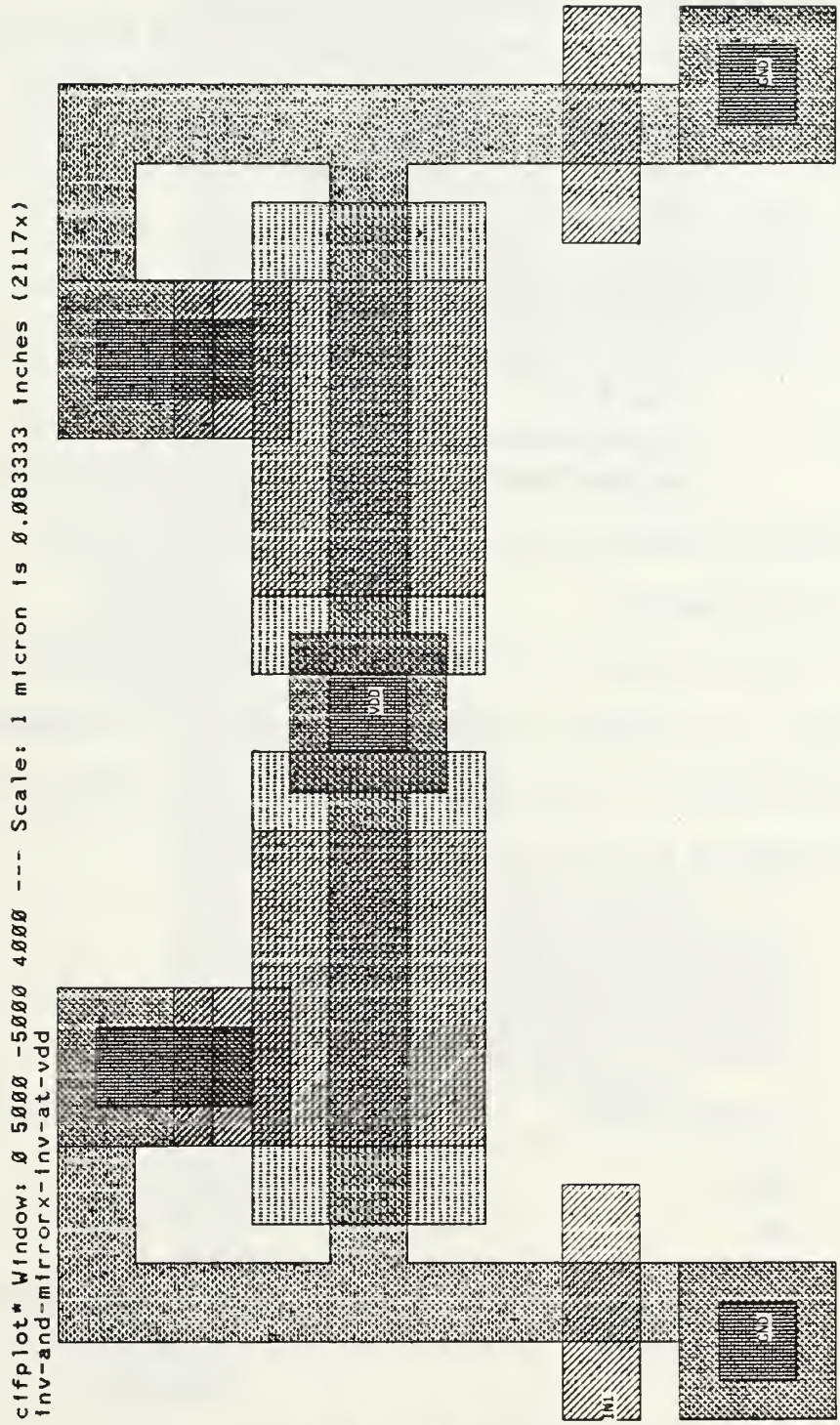

Figure 4.2 **(align-items inverter 'vdd (mirrorx inverter) 'vdd)**

```
-> (let
    ((inverter (layout-inverter 4 t)))
    (merge inverter (move inverter 20 0)) )<CR>
;; Merge the inverter and its moved image [Figure 4.3].
    (0 -20 40 0
        (((gnd)  18 -18 NM (power))
         ((in1)  14 -20 NP  (in))
         ((vdd)   8  -2 NM (power))
         ((gnd)  38 -18 NM (power))
         ((in1)  34 -20 NP  (in))
         ((vdd)  28  -2 NM (power)) )
        (1 4 6 7)
        ((symbol-call 7)
         (move (mirrorx (symbol-call 7)) 0 -4)))
```

The next operators aid moving and merging operations.

b. Query Operators

L5 has a group of operations that return an **item**'s width and length, check if an **item** is **null** and abbreviate the **defstruct** field selector functions for the MBB dimensions [e.g., **item-left** is shortened to **left**]. The argument to all these functions is an **item**:

TABLE 4.7

ITEM QUERY OPERATORS

Function	Description
is-item-null?	are the item's tree and points null?
left	same as item-left
right	same as item-right
top	same as item-top
bottom	same as item-bottom
item-width	difference between the item's right & left[11]
item-length	difference between the item's top & bottom

[11] A **null-item**'s length or width is 0. Originally L5 returned **nil** for a **null- item**. A **mark** also has 0 length and width.

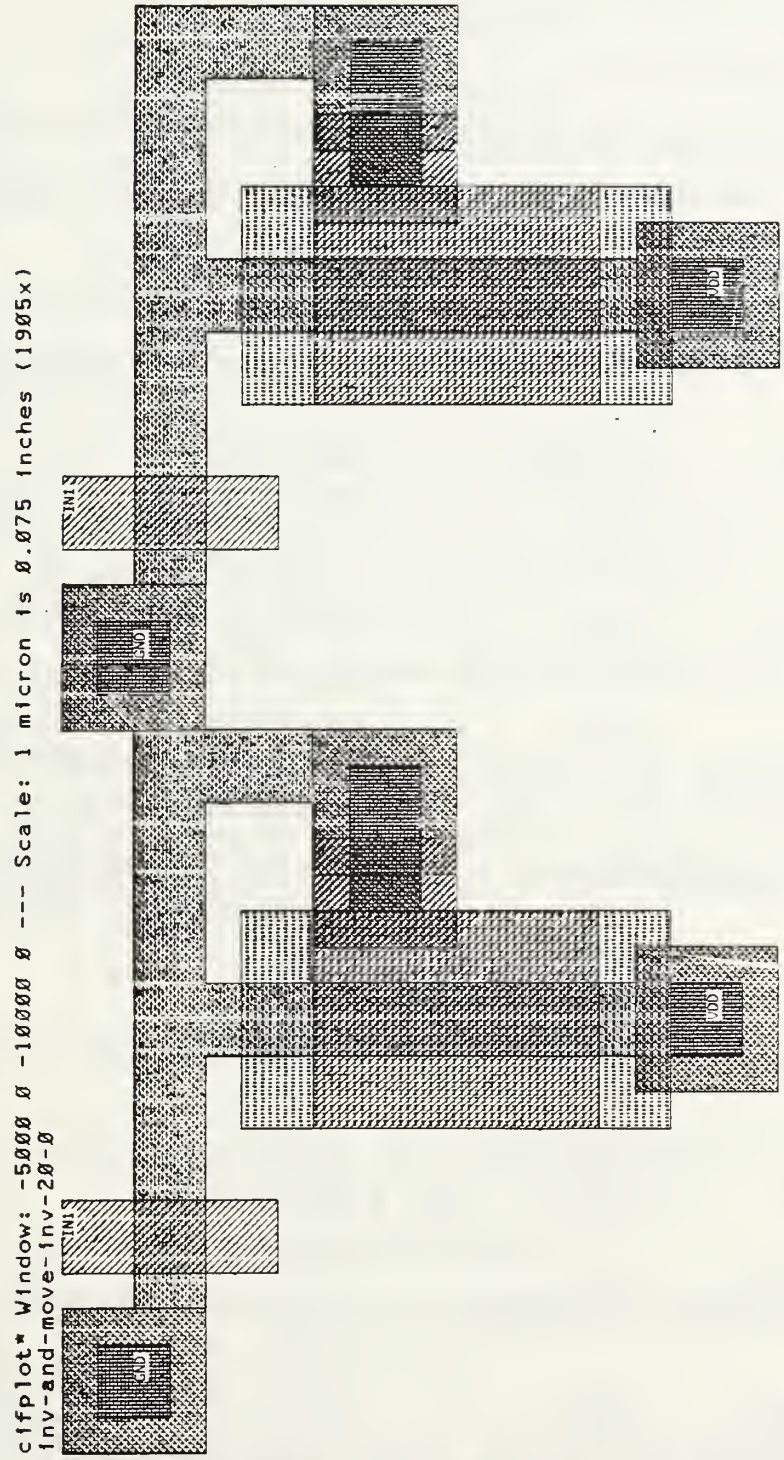

Figure 4.3 **(merge inverter (move inverter 20 0))**

An item's dimensions are useful for placement operations. Another way to access an item is by labeling it.

c. Point Operators

Labels are used by simulation, routing and timing programs. L5 has a number of functions that manipulate or use an **item**'s **points**. They are summarized in Table 4.7 below: (Crouch, 1983, pp. 11-14)

TABLE 4.7

POINT OPERATORS

Function	Arguments	Description
find	<item> <point-name>	get the first point in the item with the given name
find-all	<item> <point-name>	return all the item's points with the given name
find-attributes	<item> <attributes>	find all the item's points with given attributes as a subset of their original attributes
match-that	<thing> <list> <predicate> **&optional** <tail>	give the list elements which satisfy a predicate relation with the given "thing"
unmark	<item> <point>	take the point off the item
unmark-name	<item> <name>	delete points with the given name from the item
unmark-attributes	<item> <attributes>	remove points whose attributes contain the given attributes as a subset
unmark-attributes-list	<item> <attributes-list>	discard points that contain any element of the given attributes list

TABLE 4.7 (CONTINUED)

POINT OPERATORS

Function	Arguments	Description
contain	<item> <name>	prepend the given name to every point's name in the item

The following examples show how point operators work. The **layout-inverter** introduced in Section IV.B.2.B is again used here. This time the +5 Volt power point is extracted from the item:

```
-> (find (layout-inverter 4 t) 'vdd)<CR>
;; Find the first point named "vdd" in a layout-inverter. Refer to
;; the previous example for (layout-inverter 4 t).
    ((vdd) 8 -2 NM (power))
```

A more complex item, **layout-and**, is shown below [Figure 4.4]:

```
-> (layout-and 4 4 t)<CR>
;; Another MacPitts organelle. This one "ands" two inputs. Note
;; that the organelle calls <symbol>s1,4,6,8,9 &10. It itself is
;; <symbol>10. The list (1 4 6 8 9 10) shows the symbols.
        (0 -43 25 0
              (((gnd) 18 -18 NM (power))
               ((vdd)  8  -2 NM (power))
               ((gnd) 21 -41 NM (power))
               ((in1)  14 -43 NP  (in))
               ((in2)  19 -43 NP  (in))
               ((vdd)   8 -25 NM (power)) )
              (1 4 6 8 9 10)
              (symbol-call 10) )
```

Since this item has more than one +5 Volt power point, they can be extracted using the following procedure:

```
-> (find-all (layout-and 4 4 t) 'vdd)<CR>
;; Find all points named "vdd" in a layout-and item.
    (((vdd) 8 -2 NM (power))((vdd) 8 -25 NM (power)))
```

Figure 4.4 **(layout-and 4 4 t)**

```
-> (let
    ((test-item
        '(0 0 3 4
              (((in) 2 3 NB (external top))
               ((vdd) 1 1 NM (power left river))
               ((out) 3 3 NP (external signal top)) )
              (1 2 3)
              (symbol-call 3) ) ) )
    (find-attributes test-item '(external top)) )<CR>
;; Find all points with "external" and "top" as a subset of their
;; attributes.[12] This method uses attributes to find points.
    (((in) 2 3 NB (external top))
     ((out) 3 3 NP (external signal top)) )
```

After a point has been used it is sometimes desirable to remove it from the item. There are several functions that accomplish this. Here are two examples of how to remove one point. The first method requires that the entire point be specified as follows:

```
-> (unmark
    (layout-inverter 4 t)
    '((gnd) 18 -18 NM (power)) )<CR>
;; Remove the point from the item.
    (0 -20 20 0
        (((vdd)  8  -2 NM (power))
         ((in1)  14 -20 NP  (in)) )
        (1 4 6 7)
        (symbol-call 7) )
```

The second way to delete a point is to use its name:

---

[12] The attributes could be a list of lists instead of a list. In that case when **find-attributes** is applied the attributes parameter has to be a list of lists. If the points are of the form:

((<name>**)** <x><y><layer>**(** {**(**<attribute>*)} **))**

Then to use **find-attributes**:

(find-attributes <item> '**((**<attribute>$_A$**)**...**(**<attribute>$_L$**)))**

```
-> (unmark-name (layout-inverter 4 t) 'vdd)<CR>
;; Remove the named point from the item.
    (0 -20 20 0
        (((in1)  14 -20 NP  (in))
         ((gnd) 18 -18 NM (power)) )
        (1 4 6 7)
        (symbol-call 7) )
```

More than one point can also be removed by either using a list of attributes. The first method removes points that have *all* the specified attributes as part of their attributes. In the next example any point with both an " external " and " top " attribute is deleted, as shown below:

```
-> (let
    ((test-item
     '(0 0 3 4
        (((in) 2 3 NM ((external)(top)))
         ((vdd) 1 1 NM ((power)(left)(river)))
         ((out) 3 3 NP ((external)(signal)(top))) )
        (1 2 3)
        (symbol-call 3) ) ) )
  (unmark-attributes test-item '((external)(top))))<CR>
;; Remove any points that have "external" and "top" as part of
;; their attributes.[13]
    (0 0 3 4
        ((vdd) 1 1 NM ((power)(left)(river-router)))
        (1 2 3)
        (symbol-call 3) )
```

The second method removes points that have *any* of the specified attributes as part of their attributes. Compare the following example, in which any point with either " left " or " top " attributes is removed, with the one just presented:

---

[13] As in **find-attributes**, the attributes can either be a list of atoms or a list of lists.

```
-> (let
    ((test-item
     '(0 0 3 4
        (((in) 2 3 NM ((external)(top)))
         ((vdd) 1 1 NM ((power)(left)(river)))
         ((out) 3 3 NP ((external)(signal)(top))) )
        (1 2 3)
        (symbol-call 3) ) ) )
  (unmark-attributes-list test-item '((left)(top))))<CR>
;; Remove any points that have "left" or "top" as part of their
;; attributes. The river attribute refers to the river router.[14]
    (0 0 3 4 nil (1 2 3)(symbol-call 3))
```

Once an item has been created, it may be desirable to give all its points a common name. By doing this, point functions that use a name as an argument to search for points will find all points with the common name.

```
-> (contain (layout-and 4 4 t) 'and-1)<CR>
;; Prepend the name "and-1" to every point's name.
    (0 -43 25 0
            (((and-1 gnd) 18 -18 NM (power))
             ((and-1 vdd)  8  -2 NM (power))
             ((and-1 gnd) 21 -41 NM (power))
             ((and-1 in1) 14 -43 NP  (in))
             ((and-1 in2) 19 -43 NP  (in))
             ((and-1 vdd)  8 -25 NM (power)) )
            (1 4 6 8 9 10)
            (symbol-call 10) )
```

Labels [**point**s or **mark**s] are useful as references to direct other functions. Notice how the next function, **layout-flags**[15], gives each of its points a " river " attribute. These labeled points can then be used by the **river** function to connect them to other items.

---

[14] See Section IV.B.2.d.

[15] This function is found in the MacPitts program flags.l.

The next results, shown in Figure 4.5, are a set of four NMOS master-slave storage elements with load, write and read control lines. The labels in Figure 4.5 have been shifted in the positive x direction for legibility. Here are the four flags:

```
->(layout-flags '(ini menie mini moe) 0 138)<CR>
;; (layout-flags (<flag-name>*) <power><flag-width>)
;; <flag-width> ::=
;;   -> (flags-required-width (<flag-name>*)<power>)<CR>
;; Instantiate four flags [storage elements].
   (-25 -97 293 145
     ((((flag-write ini))      7 -97 NP (river))
      (((flag-read ini))      21 -97 NP (river))
      (((flag-load ini))      62 -97 NP (river))
      (((flag-write menie)) 67 -97 NP (river))
      (((flag-read menie))   81 -97 NP (river))
      (((flag-load menie))  122 -97 NP (river))
      (((flag-write mini))  127 -97 NP (river))
      (((flag-read mini))   141 -97 NP (river))
      (((flag-load mini))   182 -97 NP (river))
      (((flag-write moe))   187 -97 NP (river))
      (((flag-read moe))    201 -97 NP (river))
      (((flag-load moe))    242 -97 NP (river)))
      (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17)
           (symbol-call 17) )
```

In the item generated above, each point can be accessed by its name or by its attributes. In this case, all the points share a common **river** attribute, which indicates, that the **river** routing function will operate on them.

Routing operations are among the operators that make effective use of **point**s. In the following section the aforementioned **river** function, which connects two coordinate lists, is presented.

Figure 4.5 **(layout-flags '(ini menie mini moe) 0 138)**

d. River Router

L5 has a simple, but useful river router with this syntax:

(river <layer><width><stretch>**(**<y left>***)(**<y right>***))**
<layer> ::= { **NM** | **ND** | **NP** | **CM** }[16]
{ <width> | <stretch> | <y left> | <y right> } ::= <number>

The <stretch> is the amount of extra reach desired on the right side of the river. The lists of left and right y coordinates are the connection "points". Both lists should contain the same quantity of numbers. The left coordinates are connected to their respective right coordinate: that is, $\langle y\ left\rangle_N$ to $\langle y\ right\rangle_N$. The <width> is the desired width of the connecting runs.

The results are easier to show than to explain, so here is an example and its plot [Figure 4.6]:

```
-> (river 'NM 3 10 '(1 8 17 26 37) '(5 17 29 41 57))<CR>
;; Connect <y left>N to <y right>N
   (0 0 43 58 nil nil
    ((rect NM 0 0 27 2)(rect NM 27 0 30 6)
       (rect NM 27 4 43 6)
     (rect NM 0 7 21 9 )(rect NM 21 7 24 18)
       (rect NM 21 16 43 18)
     (rect NM 0 16 15 18)(rect NM 15 16 18 30)
       (rect NM 15 28 43 30)
     (rect NM 0 25 9 27 )(rect NM 9 25 12 42)
       (rect NM 9 40 43 42)
     (rect NM 0 36 8 38)(rect NM 3 36 6 58)
       (rect NM 3 56 43 58)))
```

Routing between the control unit and the data path section in MacPitts is done with **river**. In LBS, all the routing between the I/O pads

---

[16] Other layers such as **CMF** or **CMS** can be easily added by placing them into the second line of the river function in L5 [where the spacing between different runs is determined]. For example:

(assoc (layer '((NM 3)(ND 3)(NP 2)(CM 3)(CMF 3)(CMS 3))))

Figure 4.6 **(river 'NM 3 10 '(1 8 17 26 37) '(5 17 29 41 57))**

and the combinational logic unit also utilize **river**. The router can be used by assigning an attribute such as **(river)** to an item's points. The points are obtained using this attribute and one of the functions discussed in Section IV.B.2.c.. The desired points' y coordinates are then extracted and passed as lists to **river**.

A brief look has been taken at L5's data structures and operators; however, they can become untenable if a large collection of items have to be handled one by one. L5 resolves this problem by allowing abstraction of items into symbols. These symbols can in turn be used to form more complex items or symbols. At each level representational complexity is reduced, allowing L5's operators to operate efficiently.

C. HIERARCHICAL REPRESENTATION

The representation of knowledge or the structure of an organization in a hierarchical fashion is commonplace (Mead, 1980, p. 292):

> We know that human organizations use hierarchical structure to extract the greatest possible benefit from the daily activities of tens of thousands of individuals. We know that complex systems can be constructed by subdividing them into less complex systems, which are again subdivided, as many times as necessary, until the resulting systems are simple enough to construct easily . . . . The organization of real estate on the silicon surface dictates a hierarchical communication system for any devices that must support global communication.

Within this hierarchical circuit structure, the Abstraction Principle, requires that **item**s be created only once from scratch: recurring patterns should be factored out (MacLennan, 1983, p. 11) This is done in L5 with the macro **defsymbol**. (Crouch, 1983, p. 16)(Cf. Ayres, 1983, p.20)

1. Defsymbols

In order to save memory and time, L5 has a **def**ine **symbol** [**defsymbol**] macro which treats items in a fashion similar to a subroutine. The **defsymbol** macro has the following syntax:

<defsymbol-name> ::=
-> **(defsymbol** <defsymbol-name>**(**<arguments>**)**<L5 form>**)**<CR>

<L5 form> ::= { <lincoln form> | <LISP form> | <L5 form> }*

When an **item** that has been defined as a **defsymbol** is called with a set of arguments it is saved as a **symbol** on the **L5-symbol-list**. Then, if it is called again by another function with the same parameters, the **L5-symbol-list** is searched for the **symbol** representing the **item**. The position of the **symbol** in the **L5-symbol-list** is returned and placed in the **called-symbol-names** field of the **item**.

If the **defsymbol** has not been called with the given set of parameters, then a **symbol** corresponding to the **defsymbol** will be placed on the **L5-symbol-list**.

There are other effects of using a **defsymbol** that depend on whether the **L5-symbol-list**'s value is **in-memory** or **on-disk**. If it's set to **in-memory** then the **item**'s **tree** is saved as part of the **symbol** that is placed on the **L5-symbol-list**. On the other hand, if it's set to **on-disk**, then the **item**'s **tree** is not stored as part of the **symbol**: the **tree** is converted to CIF and output to the **L5-symbol-file**. These two possibilities and their effects are summarized in Figure 4.7:

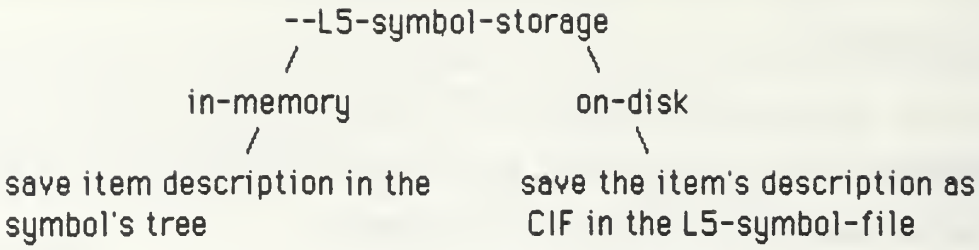

Figure 4.7 **L5-symbol-storage**

It is useful at this point to create a function to set the values of a few global variables. This is useful because during an interactive session the **L5-symbol-list** becomes very large; and, it also helps to be able to give global variables values the user typically uses. For example:

```
-> (defun start ()
     (setq --L5-symbol-list nil)
     (setq --L5-symbol-number nil)
     (setq --L5-symbol-storage 'in-memory)
     ;; patom := put atom, outputs its arguments to the given port
     ;; [it defaults to the screen].
     (patom '| The L5-symbol-list/number are nil. |)
     ;; terpri := terminate printing.
     (terpri)
     (patom '| The L5-symbol-storage is in-memory. |)
     (terpri) )<CR>
   ;; Whatever values the user desires could be set, For example:
   ;;   (minimum-feature-size! 150)
   ;;   (technology! 'cmos)
        start
```

A comparison is now made of the two ways to store symbols.

a. **in-memory** Storage

Assuming that lincoln and L5 have been loaded into LISP, the global variables can be reset as follows:

```
-> (start)<CR>
;; Reset the symbol-list/number/storage.
        The L5-symbol-list/number are nil.
        The L5-symbol-storage is in-memory.
```

The storage location has been set to **in-memory**. A look at how a **defsymbol** works is now taken:

```
-> (defsymbol butting-contact ()
    (merge
        (rect 'NM 0 -6 4  0)
        (rect 'ND  0 -4 4  0)
        (rect 'NP  0 -6 4 -3)
        (rect 'NC  1 -5 3 -1) ) )<CR>
;; A butting-contact is one of L5's defsymbols.
    butting-contact

-> (butting-contact)<CR>
;; Create a butting-contact item. Notice it calls <symbol>1 in the
;; L5-symbol-list [itself].
     (0 -6 4 0 nil (1)(symbol-call 1))
```

What happened to all the layout information in the **butting-contact**? It has been placed on the **L5-symbol-list**.

```
-> (L5-symbol-list)<CR>
;; The L5-symbol-list only contains a symbol for butting-contact.
;; symbol-ID := (butting-contact 4)
;; symbol-nest-level := 1
;; symbol-tree := ((rect NM 0 -6 4 0)...(rect NC 1 -5 3 -1))
    (((butting-contact 4) 0 -6 4 0 nil nil 1
        ((rect NM 0 -6   4   0)
         (rect ND  0 -4 -4   0)
         (rect NP  0 -6   4 -3)
         (rect NC  1 -5   3 -1))))
```

A **defsymbol** can be retrieved as a function from the **L5-symbol-list** using the **L5-item-to-program** function using this syntax:

```
(def <function-name> (lambda nil <L5 form>*)) ::=
  -> (L5-item-to-program <item form><function-name>)<CR>

<item form> ::= { <item> | <defsymbol-name> }
```

A few caveats about the use of **L5-item-to-program** are in order here:

- Avoid making the <function-name> := <defsymbol-name>. The function that's generated replaces any existing function with the same name, specifically: an item created with a **defsymbol** definition would be replaced with a zero argument function with the offending name. Consequently, when the unsuspecting user attempts to use the **defsymbol** with arguments, the system will return errors.

- Don't use **on-disk** storage. If a **defsymbol** is to be converted in this fashion, then the storage location must be **in-memory**.

These ideas are best illustrated with an example of an item that has previouly been looked at, a **(layout-inverter 4 t)**:

```
-> (L5-item-to-program
                        (layout-inverter 4 t) 'invert)<CR>
;; Make the item (layout-inverter 4 t) into a function.
   (def invert
    (lambda nil
     (merge
      (move (diff-cut) 16 -16)
      (rect 'ND  7 -18 16 -16)
      (rect 'NP13 -20 15 -14)
      (layout-pullup)
      (mark 'gnd 18 -18 'NM '(power))
      (mark 'in1  14 -20 'NP  '(in))
      (mark 'vdd  8   2 'NM '(power)))))
```

This function is a specific instantiation of the original **defsymbol**. Notice that the **defsymbol** has arguments:[17]

---

[17] The reader can take a look at a **defsymbol**'s definition using the pretty print function. The output is a function that calls up two subsidiary functions, **desymbol1** and **defsymbol2**. These functions check to see if an item is already on the **L5-symbol-list** and if not, they add a new symbol to it. For example, try: **-> (pp butting-contact)**

```
-> (defsymbol layout-inverter (ratio1 mark?)
    (let
     ((diffusion
      (merge
       (move (diff-cut) 16 -16)
       (mark 'gnd 18 -18 'NM '(power))
       (cond
        ((= ratio1 4)(rect 'ND 7 -18 16 -16))
        (t (rect 'ND 7 -20 16 =16)))))
      (gate (rect 'NP 13 -20 15 -14))
      (mark
       (cond
        (mark? (mark 'in1 14 -20 'NP '(in)))
        (t (null-item)))))
     (merge diffusion gate mark (layout-pullup))))<CR>
;; The desymbol macro returns a name.
       layout-inverter
```

If many large items are placed on the **L5-symbol-list** and all their trees are also placed there, the list quickly becomes unwieldy. An alternative is to keep all the other information in the **L5-symbol-list**, convert the item's tree to CIF and place the CIF in the **L5-symbol-file**.

b. **on-disk** Storage

This storage mode reduces the **L5-symbol-list**s size by changing the item's tree to CIF. It is useful when items don't need to be retrieved from the **L5-symbol-list** with their trees [for example, the **L5-item-to-program** function will only create a program out of a symbol if its tree is on the **L5-symbol-list**; similary, the Caesar conversion routines {Section IV.C.3} also require **in-memory** storage].

An example can be examined after resetting all the global variables using the **start** function. First the **L5-symbol-storage** is set to **on-disk** and the effect compared with the results of Section IV.C.1.a.

-> (start)<CR>
The L5-symbol-list/number are nil.
The L5-symbol-storage is in-memory

-> (L5-symbol-storage! 'on-disk)<CR>
;; Set the symbol storage location to on-disk
on-disk

-> (butting-contact)<CR>
;; The butting contact item looks the same as before.
(0 -6 4 0 nil (1)(symbol-call 1))

-> (L5-symbol-list)<CR>
;; Although the butting-contact item is the same the <symbol> no
;; longer has a tree in it. Compare to the L5-symbol-list in the
;; previous section.
(((butting-contact 4) 0 -6 4 0 nil nil 1))

In this case the L5-symbol-list contains only one symbol. If there are several symbols in it, then they can be individually retrieved in item form as follows:

<item> ::= **-> (create-called-symbol-item** <position>**)**<CR>
<position> ::= <integer>

Therefore, continuing with the above example:

-> (create-called-symbol-item 1)<CR>
;; The item that is returned has a symbol-call tree. This means
;; that the item has been output as CIF[18] and is referred to as CIF
;; symbol 1.
(0 -6 4 0 nil (1) (symbol-call 1))

This is exactly the result that evaluating **(butting-contact)** gave. The price gained in speed and storage is the less descriptive format. However, the item has been converted to CIF.

---

[18] See Footnote 8 of this chapter. Section IV.C.2 covers this in more detail.

2. CIF

When the **on-disk** storage mode is used, an item's tree is sent to the **L5-symbol-file** as CIF. A brief look is now taken at the CIF result from the previous example:

-> (L5-symbol-file)<CR>
/tmp/L5sym20374

-> (exec cat /tmp/L5sym20374)<CR>
;; exec allows UNIX® functions to be performed from LISP. In this
;; case the contents of the L5-symbol-file are concatenated to
;; the terminal. The following is CIF output. CIF uses " (" and ") "
;; for comments.

DS 1;
(define <CIF symbol> named 1);
(name: butting-contact);
(CIF comments are not printed out);
L NM; B L 1000 W 1500 C 500, -750;
(since this was output with 250 centimicrons = lambda);
(all the units must be divided by this amount);
(CIF defines its rectangles like L5 does its boxes:);
(<layer><length><width><x_{center}><y_{center}>);
((box 'NM 4 6 2 -3));
L ND; B L 1000 W 1000 C 500, -500;
((box 'ND 4 4 2 -2));
L NP; B L 1000 W 750 C 500, -1125;
((box 'NM 4 3 2 -3));
L NC; B L 500 W 1000 C 500, -750;
((box 'NM 2 4 2 -3));
DF;
(end <CIF symbol>$_1$'s definition);

The function that L5 uses to create CIF has the following format:

<file>**.cif** ::= **-> (cifout** {[']<item>}{[']<file>}{"<title>"}**)**<CR>

A few comments on using this function are:

- The number of centi-μ/λ should be set using **minimum-feature-size!** before outputting CIF. If an item was created with on-disk storage, then the CIF has already been made at whatever **minimum-feature-size** the system is currently using. Therefore, functions such as **cifplot** must be given the scale to use if it is not 2.0 centi-μ/λ; otherwise, plotting errors will occur.

- The title appears as a CIF comment. For example, if the title is: **"butting-contact"**,
then the CIF comment line is:
(Title : butting-contact);

- If points or marks are to appear as CIF they should have the **external** attribute. The conventional " 94 " CIF user extension for labels has been added as a default option to the function **cifout-external-name** in L5.1 to enable the standard cifplot program to plot marks and points. (Carlson, 1984, p.78)

Although L5 items can be transformed into CIF, the inverse function is not directly implemented in L5. In order to accomplish this, Caesar format is used.

3. Caesar

L5 is used to design circuits in a procedurally oriented approach. Another, and perhaps the most widely used VLSI design methodology, is graphical. L5 provides access to the interactive graphical editor Caesar. At the present time, the Caesar editor can be invoked from the L5 environment. Conversely, Caesar files can be changed to L5 format. Figure 4.8 shows the different ways to get from one format to the other:

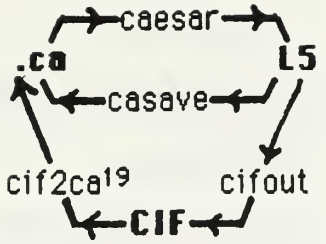

Figure 4.8 Caesar, L5 and CIF Conversions

There are several functions associated with the Caesar editor. Table 4.8 gives a synopsis (Crouch, 1983, pp. 17-18):

TABLE 4.8

CAESAR FUNCTIONS

Function	Arguments[20]	Description
caesar	<item><file>	Converts an <item> into Caesar format, the Caesar editor is invoked and the results of the session are saved in the <file> as items.
display	<item>	Displays the <item> in Caesar without generating any L5 code afterwards.
caout	<item><file>	Outputs the <item> in Caesar format to <file>**.ca**
cain	<file>	Reads in a Caesar formatted file and converts it to L5 code.
casave	<caesar file> <L5 file>	Converts a <caesar file> into L5 code and saves the result in <L5 file>. Each Caesar symbol is made into an L5 **defsymbol**.

---

[19] This is a Berkeley CAD conversion program from CIF to Caesar format. A minor problem with this present scheme is that Caesar evolved into the more versatile Magic system. The routines need to be modified to use Magic format instead of Caesar format.

[20] All of these functions are fexprs, therefore, none of the arguments are quoted.

TABLE 4.8

CAESAR FUNCTIONS (CONTINUED)

Function	Arguments	Description
cadef	<caesar file>	Places each Caesar symbol on the **L5-symbol-list**.

4. Summary

This chapter has covered the significant features of L5. Its hierarchical style is well suited for writing procedures which manipulate lower level primitives. These basic building blocks can be designed on a graphical layout editor and then converted into L5 programs.

At the present time L5 uses the Caesar editor. In the future a routine should be implemented to use the more versatile Magic editor.

V. TOP.L AND PREPASS.L: THE TOP-LEVEL

The reader has seen how a group of LISP object files can be loaded together to create a LISP environment[1]. This chapter shows how a top-level function is used to make an environment accessible with parameters. In other words, the environment will be invoked just like other UNIX® commands. Two programs that contain top-level functions are top.l [LBS] and prepass.l [MacPitts]. In addition, these programs contain the "compilers" for LBS and MacPitts. A look at these top-level programs brings to light major differences and similarities between LBS and MacPitts.

A. THE TOP-LEVEL

1. Franz Lisp's Default Top-Level

A top-level function creates the prompt-read-eval-print loop.The user can call the top-level function and can create a prompt-read-eval-print loop with different characteristics. To do this, the user defines a new top-level function and types **(reset)** to run it. (Foderado, 1983, p.13-1)(Wilensky, 1984, p.138)

When the imperative command **lisp** is given to UNIX®, the interpreter is brought into action with its default top-level: **franz-top-level**[2]. This occurs because the variable **top-level** is bound to **franz-top-level**.

---

[1] See the discussion in Chapter II Section B.2.b.

[2] Defined in /usr/lib/lisp/toplevel.l

Some of the actions taken by this default top-level are (Foderado, 1983, Appendix C.2)(Wilensky, 1984, p. 348):

- Print out: **Franz Lisp Opus 38.69**

- Load .lisprc from the user's HOME directory.

- Enter a prompt-read-eval-print loop. Each time through, before the prompt is printed, the value of **user-top-level** is checked. If it has been bound, then its value will be **funcall**ed. This allows the user to enter a different top-level for the interpreter. Handle errors, interrupts, etc..

2. Example Top-level

At this point, it's useful to use a simple makefile and top-level file to examine the ideas involved in more detail. The makefile is as follows:

```
chip: L5.o defstructs.o top.o creator.o
      (echo "\
       (eval-when (eval)\
        (sstatus translink on)[3]\
        (fasl 'lincoln)\
        (fasl 'L5)\
        (fasl 'defstructs)\
        (fasl 'top)\
        (fasl 'creator)\
        (setq user-top-level 'chip-top-level)\
        (signal 2 'chip-interrupt-handler)\
        (setq option-list '(stat obj cif))\
        (minimum-feature-size! 150)\
        (dumplisp chip)\
        (exit))" | lisp
```

---

[3] Makes LISP transfer of control to compiled functions fast (Wilensky, 1984, p. 284). This may be a disadvantage if debugging and tracing is to be done within this dumplisp environment, since the **trace** and **debug** functions will not work. If the primary intent is to debug code then use: **(sstatus translink nil)**

The top-level file, top.l, is composed of several LISP functions. The first, **chip-top-level**, performs a check of the arguments used when invoking "chip". If there are no arguments then the "chip" dumplisp environment is called up. If there are arguments, then these are passed on to the **chip-compiler** function. The user should note that **chip-top-level** was set to be the top-level function in the example's makefile above.

In other words, the "chip" dumplisp environment has a function, **chip-top-level**, which handles the arguments placed in the command line when "chip" is invoked. Notice that if no arguments are given, then a message is printed out and the user is placed into the " chip " dumplisp environment. This feature can be used for debugging purposes [See Footnote 3 of this chapter]. A look at this function follows:

```
(defun chip-top-level ()
;; If "chip" is invoked without any arguments:
  (cond ((=1 (argv))
  ;; (argv) gives the number of elements on the command
  ;; line that invoked this LISP. So, if the user types:
  ;;      % chip <argument1><argument2><argument3>
  ;;  then (argv) := 4
          (patom
              "usage: chip <filename> [<options>]")
          ;; (patom <expression>[<port: default to screen>])
          ;; print out the expression:
          ;;     usage: chip <filename> [<options>]
          ;; The " [ " and " ] " indicate an optional argument.
          (terpr)
          ;; (terpr) or (terpri) terminates printing.
          (setq user-top-level ())
          ;; The variable user-top-level is set to nil, but
          ;; notice that in the makefile it was set to chip
          ;; top-level. Therefore, if chip is called up
          ;; without arguments, then chip-top-level calls
          ;; up the " chip " dumplisp environment
```

```
;; and gives control to franz-top-level, e.g.:
;;  % chip
;;  usage: chip <filename>[<options>]
;;
;;  [Return to top level]
;;  ->
(signal 2 (function sys:int-serv))
;; (signal <number><function>)
;; The number 2 represents an interrupt signal
;; and sys:int-serv is the LISP default function
;; that handles interrupts [See I.B.1.c]. (Wilensky,
;; 1984, pp. 269 & 355)(Foderado, 1983, p. 6-7)
(reset)
;; If "chip" is invoked with arguments:
(t
  (chip-compiler
  ;; (chip-compiler <file-name> [<options>])
    (mapcar
    ;; The mapping function mapcar retrieves the
    ;; arguments typed in the command line, e.g.:
    ;;      % chip multiplier cif obj
    ;; then,
    ;; (mapcar (lambda (x)(argv x))(1 2 3))
    ;; returns:
    ;; (multiplier cif obj)
      (lambda (index)(argv index))
      ;; (argv <index-number>)
      ;; returns the indexed argument on the
      ;; command line, e.g. If,
      ;;     % chip adder cif
      ;;  then,
      ;; (argv 1) := adder and (argv 2) := cif
      ;;      -> (argv 0)
      ;;          chip
      ;; and (argv) gives the total number of
      ;; arguments on the command line: 3
      (count (1- (argv))) ) ) ) )
      ;; (count <integer>)
      ;; make a list of integers up to the given
      ;; one.
```

```
;;      -> (count 5)
;;           (1 2 3 4 5)
(exit) )
```

The next function is set up in the makefile to handle interrupts. When an iterrupt is received it prints out "**chip-interrupt:**", the signal number and then exits the "chip" dumplisp environment. Here is the code:

```
(defun chip-interrupt-handler (signal-number)
;; This is used in the makefile as the function that
;; handles interrupts (2), floating exceptions (8),
;; alarms (14) and hang-ups (1). (Wilensky, 1984, p. 270)
  (patom " chip-interrupt: ")
  (patom signal-number)
  (terpr)
  (exit)) )
```

The previous functions allowed the user to invoke the "chip" dumplisp environment as a UNIX® command. The following function is used within the "chip" dumplisp environment to pass arguments to the **chip-compiler** function:

```
(def chip
 ;; The nlambda function format takes many arguments,
 ;; they are unevaluated and bound as a list to the
 ;; function's single parameter. For example:
 ;;     -> (chip adder cif obj)
 ;; then args := (chip adder cif obj)
  (nlambda (args)
   ;; (chip <filename> [<option>*])
   ;; <option> ::= {nostat | noobj | nocif | mag}
   ;; <default option> ::= stat obj cif
    (chip-compiler args) ) )
```

The next function coordinates other programs in order to produce the different types of output. It first uses the **process-option** function to set the global variable, **option-list**, to the options that have been input. Then

it calls on other programs to process the options and place the outputs in appropriately labeled files.

```
(defun chip-compiler (args)
;; (chip-compiler '(<filename>[<option>*]))
 (cond
  ((not (null args))
   ;; If there are arguments then find the options using
   ;; the process-option function. This will place all
   ;; requested options along with other defaults which
   ;; were not inhibited on the option-list.
    (mapcar 'process-option (cdr args))
    (prog (in-file stat-file mag-file obj-file
           out-file inport statport magport
           objport netlist output)
     ;; After declaring all the prog's local variables,
     ;; set the filenames [e.g., <filename>.chip,
     ;; <filename>.stat, etc..].
     (setq in-file (concat (car args) '.chip)
     (setq stat-file (concat (car args) '.stat)
     (setq mag-file (concat (car args) '.mag)
     (setq obj-file (concat (car args) '.obj)
     (cond
     ;; If the in-file exists, then take the following
     ;; actions.
      ((probef in-file)
       (setq inport (infile in-file))
       (setq statport (fileopen stat-file 'w))
       (setq netlist
          (converter (read inport) stat-port))
       (cond
        . ;; If "obj" is in the option list, then . . .
          ((member 'obj option-list)
          { & other function calls to other
            subprograms as required to make
            the chip} ] )
```

The **option-list** has certain default values set in the makefile. The user can inhibit them by placing "no" in front of them [e.g. **nostat**]. On the

other hand, there may be other options, besides the default values, which the user can input. The following function examines the options the user has input and updates the option-list as required:

```
(defun process-option (option)
 (cond
  ((not (atom? option))
   (warning "Option not atom"))
    ;; Options must be atoms.
  ((and (> (length (explode option)) 2)
        (equal 'n (car (explode option)))
        (equal 'o (cadr (explode option))) )
   ;; Is the option more than two letters long and its
   ;; first two letters an "n" and "o" [the option is of the
   ;;  form: noXXX]? Explode separates an atom into the
   ;; characters that compose it (implode is its dual).
   (cond
    ;; Is the rest of the option [i.e. excluding the " no "]
    ;;  in the option list?
     ((member?
       (implode (cddr (explode option)))
       option-list )
      ;; Drop the option from the option-list.
      (setq option-list
            ;; Remove the indexed element from the
            ;; option-list.
           (nthdrop
            ;; Find the index of the option without the
            ;; " no " in the option-list.
            (iota
              (implode (cddr (explode option)))
              option-list )
            option-list ) ) ) )
   ;; Otherwise, if the option is not of the form noXXX,
   ;; then add it to the option list.
   (t (cond ((not (member? option option-list))
            (setq
             option-list
             (cons option option-list) ) ) ) ) ) )
```

Both LBS and MacPitts have top-level functions similar to these. The major difference between the two programs is that LBS generates combinational logic circuits, whereas MacPitts has a control and data-path paradigm. A look at their compilers illustrates their differences.

B. LBS COMPILER

LBS accepts a file with LISP boolean forms as its input. These boolean forms have the following syntax:

TABLE 5.1

LBS SYNTAX

Category	Syntax
<LBS program> ::=	**(**<bform>*)**
<bform> ::=	{ **(name** <name>**)** | **(setq** <name><bform>**)** | **(out** <name><bform>**)** | **(nor** <bform>***)** | **(or** <bform>***)** | **(and** <bform>***)** | **(not** <bform>***)** | **(xor** <bform>***)** }[4]

So for example, an LBS program might consist of the following lines:

```
((setq  load (nor (and (not in1) in2 clock1) in3)
 (out   out1 (xor (nand in1 clock2) in4 load)) )
```

If a <name> is used as the first argument of a **setq** or an **out** it can be used in other <bform>s. Otherwise, the other <name>s are assumed to be inputs. The input file is parsed and converted into an intermediate "obj" format by **bool-to-straps** [located in extract.l]. This is a connectivity list that is used to generate the Weinberger array implementation of the boolean

---

[4] Of course, **bform** is implemented as a **defstruct**.

expressions. This "obj" format is then used by **layout-cmos-wein** [called by **layout-chip**] or **layout-inside** [used in the Caesar section] to create the array. These ideas can be seen in the implementation of LBS's compiler. The **lbs-compiler** function assumes that an **lbs-top-level**, **lbs-interrupt-handler** and **lbs** function are available [See Section V.A]. LBS's compiler has the following format:

```
(defun lbs-compiler (args)
 (cond
  ;; If the arguments aren't empty, then process them.
  ((not (null args))
   (mapcar 'process-option (cdr args) )
   (prog
    ;; Define local variables.
    (in-file stat-file caes-file obj-file out-file
     inport statport caesport objport bs chip )
    (setq in-file (concat (car args) '.lbs))
    (setq stat-file (concat (car args) '.stat))
    (setq caes-file (concat (car args) '.ca))
    (setq obj-file (concat (car args) '.obj))
    (setq out-file (car args))
    ;; Check that the input file is not empty and then pro-
    ;; proceed to process the input file.
    (cond
     ;; Probe the input file to see if it has anything in it.
     ;;  If it's not empty then turn it into the in port.
     ((probef in-file)
      (setq inport (infile in-file))
      (setq statport (fileopen stat-file 'w))
       ;; The boolean input format is converted to a
       ;; format showing connectivity and logical
       ;; relationships.
      (setq
        bs
        (bool-to-straps (read inport) statport) )
      ;; Check the options and produce the ones desired.
      (cond
          ;; Produce the intermediate "obj" format.
```

```
    ((member 'obj option-list)
      (setq objport (fileopen obj-file 'w))
      (pp-form bs objport)
      (terpr objport)
      (close objport)
      (message-number-7) )
     ;; Produce Caesar format output.
    ((member 'hca option-list)
      (message-number-5)
      (setq caesport (fileopen caes-file 'w))
      ;; Create the Weinberger array.
      (setq
        chip
        (layout-inside
          bs
          (car args)
          caesport) )
       ;; Create CIF format output.
      (cond
        ((null chip)(message-number-4))
        (t (cifout chip out-file out-file)
           (message-number-6) ) ) )
     ;; Create CIF format output.
    ((member 'cif option-list)
     (message-number-1)
      ;; Build a Weinberger array then route it to the
      ;;  I/O pads.
     (setq chip (layout-chip bs))
     (cond
       ((null chip)(message-number-4))
          (t (cifout chip out-file out-file)
          (message-number-2) ) ) ) )
 ;; Make a simulation file.
(cond
 ((member 'sim option-list)
  (sim bs out-file))))
 (t (message-number-3 in-file)) )
(return) ) ) ) )
```

LBS has a simple architecture based on implementing combinational logic circuits in CMOS. MacPitts is a larger program with many more possibilities.

C. MACPITTS[5] COMPILER

The increase in complexity from LBS to MacPitts can easily be seen in the syntax used by the latter program. A glance through MacPitt's BNF shows that it incorporates concepts such as: **function**, **macro**, **port** [n-bit data], **signal** [t or f data], **register** [datapath storage], **flag** [signal storage], **organelle** [functional unit], **test** [e.g., = or =**0**], etc.. MacPitts, unlike LBS, requires that I/O pads be specifically declared [that's why <pin-number>s are used to specify their location]. Again it should be noted that most of these ideas are implemented as **defstruct**s.[6] Skim through the BNF to gain a feeling for MacPitts' syntax:

TABLE 5.2

MACPITTS SYNTAX

Category	Syntax
<MacPitts program> ::=	**(program** <program-name><word-size> {<eval> | <def> | <always> | <process>}+**)**
<eval> ::=	**(eval** {**compile** | **simulate** | **both**} <LISP form>**)**
<def> ::=	**(def** <pin-number> { **power** | **ground** | **phia** | **phib** | **phic**}**)**
<def> ::=	**(def** <register-name> **register)**

---

[5] Only a brief description is given here of MacPitts. The reader should consult Southard [RVLI-3], 1983, pp. 1-33 and Siskind, 1981, pp. 1-18.

[6] This will be covered in more detail in Chapter VI.

TABLE 5.2 (CONTINUED)

MACPITTS SYNTAX

Category	Syntax
<def> ::=	**(def** <port-name> **port** {**input** | **output** | **tristate** | **i/o**} **(**<pin-number>$^+$**))**
<def> ::=	**(def** <port-name> **port internal)** |
<def> ::=	**(def** <flag-name> **flag)**
<def> ::=	**(def** <signal-name> **signal** {**input** | **output** | **tri-state** | **i/o**} <pin-number>**)**
<def> ::=	**(def** <signal-name> **signal internal)**
<def> ::=	**(def** <constant-name> **constant** <form>**)**
<def> ::=	**(def** <organelle-name> **organelle** <#control-lines><#parameters><#test-lines> <result?><GEN form><SIM form>**)**
<def> ::=	**(def** <function-name> **function** <organelle-name> **(**{**integer** | **boolean**}$^+$**)** **(**<control-line>***)(**<parameter>$^+$**)**<INT form>$^+$**)**
<def> ::=	**(def** <test-name> **test** <organelle-name> **(**{**integer** | **boolean**}$^+$**)(**<control-line>***)** **(**<parameter>$^+$**)**<test-line><INT form>**)**
<def> ::=	**(def** <macro-name> **macro** {**single** | **list**} <LISP form>$^+$**)**
<pin-number> ::=	<integer>
<result?> ::=	{**yes** | **no**}
<always> ::=	**(always** <form>$^+$**)**
<process> ::=	**(process** <process-name> <stack-depth>{<label> | <form>}***)** | <macro>
<stack-depth> ::=	<integer>
<label> ::=	<symbol>

TABLE 5.2 (CONTINUED)

MACPITTS SYNTAX

Category	Syntax
<form> ::=	<integer> | "<character>" | <constant-name> | <register-name> | <port-name> | **(go** <form>**)** | **(call** <form>**)** | **(return)** | **(par** <form>*****) | **(if** <form><form><form>**)** | **(cond** {**(**<condition><form>$^{+}$**)**}$^{+}$**)** | **(setq** <register-name><form>**)** | **(setq** <port-name><form>**)** | **(setq** <signal-name><condition>**)** | **(**<function-name><formula>$^{+}$**)** | <macro>
<macro> ::=	**(macro** <macro-name>{**single** | **list**} <LISP form>$^{+}$**)**
<organelle> ::=	**(organelle** <organelle-name> <*control-lines><*parameters><*test-lines> <result?><GEN form><SIM form>**)**
<function> ::=	**(function** <function-name> <organelle-name> **(**{**integer** | **boolean**}$^{+}$**)** **(**<control-line>*****)**(**<parameter>$^{+}$**)**<INT form>$^{+}$**)**
<test> ::=	**(test** <test-name><organelle-name> **(**{**integer** | **boolean**}$^{+}$**)**(**<control-line>*****) **(**<parameter>$^{+}$**)**<test-line><INT form>**)**
<condition> ::=	**t** | **()** | <signal-name> | **(and** <condition>$^{+}$**)** | **(or** <condition>$^{+}$**)** | **(not** <condition>**)** | **(nor** <condition>$^{+}$**)** | **(nand** <condition>$^{+}$**)** | **(xor** <condition>$^{+}$**)** | **(equ** <condition>$^{+}$**)** | **(bit** <bit*>{<integer>|<integer form>}**)** | **(setq** <signal-name><condition>**)** | **(**<test-name><form>$^{+}$**)** | <macro>

TABLE 5.2 (CONTINUED)

MACPITTS SYNTAX

Category	Syntax
<???-name> ::=	<symbol>

The increased syntactical complexity is handled by functions in prepass.l that parse through the input forms. They can be divided into three major categories: functions that fetch something, those that manipulate data, and those that expand forms. They are used in conjunction with other programs in MacPitts and the **defstruct** macro to generate an intermediate "obj" description[7]. This "obj" code can then be implemented in a controller and data path structure. The functions have the following syntax and in general their names are an indication of what they do:

TABLE 5.3

PREPASS.L FUNCTION SYNTAX

Function	Syntax
get-<x>	
<x> ::=	{ **source** | **library** | **object** | **program-name** | **program-tail** | **word-length** | **definitions** | **sources** | **sources-from-form-list** | **sources-from-form** | **destinations** | **destinations-from-component-list** | **destinations-from-form-list** | **destinations-from-form** | **labels-from-component-list** | **labels-from-form-list** }

---

[7] This format is more complex than LBS's "obj" code. MacPitts "obj" format can be broken up into five sections: definitions, flags, data-path, control and pins. LBS only has a control section.

TABLE 5.3

PREPASS.L FUNCTION SYNTAX

Function	Syntax
process-<y>	
<y> ::=	{ <z> | **definitions** | **definition** | **control-line** | **parameter** | **test-line** }
<z> ::=	<zz>-**definition**
<zz> ::=	{ **power** | **ground** | **phia** | **phib** | **phic** | **register** | **flag** | **signal** | **port** | **macro** | **constant** | **organelle** | **function** | **test** }
expand-<YY>	
<YY> ::=	{ **process** | **macro** | **form-list** | **form** | **component-list** | **component** }

A quick look is now taken at MacPitts' compiler. It works in a fashion similar to the LBS compiler. First, the input forms are converted to "obj" format, and then this object code is transformed into the requested options. Here is the compiler function:

```
(defun macpitts-compiler (operands)
 (prog (file-name file object obj item)
   ;; ptime gives run and garbage collection times.
   (setq initial-ptime (ptime))
   ;; The number of garbage collections that occurred.
   (setq initial-gccount $gccount$)
   ;; If the operands are null or atoms then return to the
   ;; franz-lisp top level.
   (cond ((or (not (list? operands))
              (null operands)
              (not (atom? (car operands))))
          (patom "usage: (macpitts <filename>
            [<options>])")
          (terpr)
          (return ())))
   (setq file-name (car operands))
   ;; Set the option-list to the requested and uninhibited
   ;; default settings.
```

```
(mapcar 'process-option (cdr operands))
(statistic (concat "for project " file-name))
(statistic (concat "options: "
                   (slash
                    (explode option-list)
                    "
                    (function concat) ) ) )
;; Set the process parameters to 5µ or 4µ. Note that
;; MacPitts' makefile defaults to 250 centi-µ/λ.
(cond
 ((member? '5u option-list)
  (minimum-feature-size! 250)))
(cond
 ((member? '4u option-list)
  (minimum-feature-size! 200)))
;; Is interpreter output one of the options?
(cond
 ((member? 'int option-list)
  (cond
   ((null (catch (interpret file-name) note))
    (return ()) ) ) ) )
;; Is CIF or object format desired?
(cond
  ((or (member? 'obj option-list)
       (member? 'cif option-list) )
   (setq object (get-object file-name)) )
  (t (return t)) )
;; If nothing is generated up to now, exit.
(cond ((null object) (return ())))
;; Output data-path statistics.
(statistic (concat "Data-path has "
                   (length
                    (object-data-path object) )
                   " Units"))
;; Is object code desired?
(cond
 ((member? 'obj option-list)
  (herald "Outputing .obj file")
  ;; Object code is made up of defs, flags, data-path,
  ;; control and pins.
```

```
    (setq obj
     (make-object
      (purge-library
       (object-definitions object) )
      (object-flags object)
      (object-data-path object)
      (object-control object)
      (object-pins object) ) )
    (setq file
     (outfile (concat file-name ".obj")) )
    (pp-form obj file)
    (close file) ) )
  ;; Was CIF desired?
  (cond
   ((member? 'cif option-list)
    (setq item
     (catch (layout-object object) note) )
    (cond ((null item)(return ())) )
    (herald "Outputing .cif file")
    (cifout item file-name file-name) ) )
  (statistic (concat "Memory used - "
             (/ (memory) 1024) "K"))
  (statistic (concat
      "Compilation took "
      (quotient
       (- (car (ptime))(car initial-ptime))
       3600.0 ) " CPU minutes" ) )
  (statistic (concat
     "Garbage collection took "
     (quotient
      (- (cadr (ptime))(cadr initial-ptime))
      3600.0 ) " CPU minutes" ) )
     (statistic (concat
     "For a total of "
     (- $gccount$ initial-gccount)
     " garbage collections") )
  (return t) ) )
```

In summary, a bird's eye view of LBS's and MacPitts' compilers shows the relative differences between the two programs. MacPitts has a more

extensive syntax which requires more parsing of input code. Though they differ in complexity, these programs share many common features. They both use a top-level function with an interrupt handler and option process to access a dumped lisp environment created with a makefile. After parsing through their input forms, they generate an intermediate object format. And finally, both use L5; and, all their data types are handled by the **defstruct** facility (including many of their inputs).

VI. ORGANELLES

Chapter V contains MacPitts' syntax and its top-level function. MacPitts' BNF allows the user to define functions, macros, tests and organelles and use them when writing a MacPitts program. Alternatively, the user can modify the organelles.l and library programs and remake MacPitts. In this fashion the new operators become part of MacPitts' syntax.

A. OVERVIEW

Before showing an example of the changes that are made in MacPitts to change its syntax, the relationships among some of its programs and functions need to be pointed out. When the user inputs a <MacPitts program>, the compiler [located in prepass.l] parses through the <MacPitts form>s. The program uses its **get**-<x>, **process**-<y> and **expand**-<yy>[1] functions to process <definition>s; evaluate <eval>s; expand <macro>s; and, obtain <source>s, <destination>s and <label>s. This is done by using list selectors to disassemble the <MacPitts program> while checking the syntactic labels that were used. For example, the words **def**, **ground**, **process**, **macro**, **function**, etc., all trigger the use of the **process-definition** function. The **eval** and **process** labels are treated separately. The functions used during the parsing process to obtain <definition>s from the input are shown below:

---

[1] Refer to Section V.C.

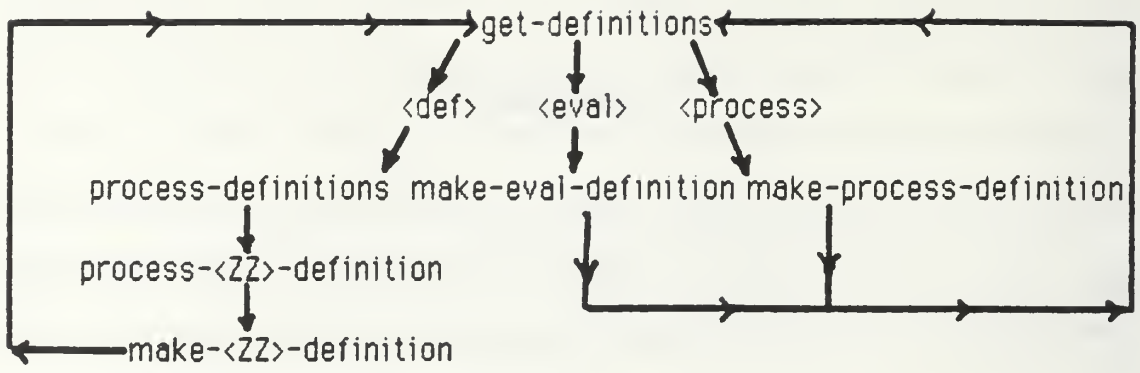

Figure 6.1 Prepass Function Flow

Once the definitions have been obtained and processed, the <flag>s, <data path>, <control>, <pin>s and <sequencer>s are **extracted** [functions located in extract.l] from them . The <control>, <flag> and <data path> elements are **framed** [layout functions in frame.l] together from the results of the **data-path**, **controller**, and **flags** generators. A separate portion of MacPitts [general.l] contains **general** query and lookup functions used by the extractor and data-path generator.

The physical layout descriptions, **organelles**, are in two sections: a compiled portion and a non-compiled program called a **library**. The data-path creator uses the compiled organelles and the extractor uses the library.

The overall program hierarchy is sketched below:

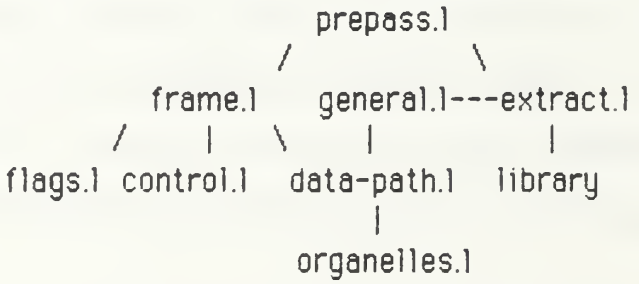

Figure 6.2 MacPitts Program Hierarchy

Prepass.l is coordinates the conversion of the <MacPitts program> to a layout with its **get-object** function. This operator uses subsidiary programs [primarily extract.l] to produce an intermediate result, an " .obj " file, which can then instantiated into the silicon mask level by layout functions [in flags.l, control.l and data-path.l]. This object file is a **defstruct** with the following definition:

```
(defstruct object
    (definitions flags data-path control pins))
```

A quick look is now taken at the names of the major functional categories in prepass.l and its helper programs. Skimming through the function names provides a " feeling " for MacPitts. The most common operators are summarized by program in Table 6.1.

TABLE 6.1

MACPITTS PROGRAM FUNCTION SUMMARY

Program	Function Name Format
prepass.l	**get**-<x>, **process**-<y>, **expand**-<zz>
extract.l	**extract**-<A> <A> ::= { **component-list** | **process** | **form** | **atom** | **list** | **string** | **fixnum** | **register** | **flag** | **port** | **signal** | **label** | **go** | **call** | **return** | etc. }
frame.l	**layout**-<B> <B> ::= { **object** | **skeleton** | **wing** | **net** | **pins** | **power-ring**}
control.l	**layout**-<C> <C> ::= { **control** | **driver** | **mpx** | **register** | **weinberger**-<D> | etc. } <D> ::= { **gates** | **nor** | **nor-inport** | **nor-gnd-line** | etc. }
data-path.l	**layout**-<E> <E> ::= { **data-path** | **buses** | **unit** | **organelle** | etc. }

TABLE 6.1 (CONTINUED)

MACPITTS PROGRAM FUNCTION SUMMARY

Program	Function Name Format
flags.l	**layout**-<F> <F> ::= { **flags** | **flags**-{ **power** | **clock**} | etc. }
general.l	**is**-<G>**?** <G> ::= { **register** | **flag** | **port** | **signal** | **macro** | **function** | **test** | **label** | etc. } **lookup**-<H> <H> ::= { **macro** | **constant** | **signal** | **port** | **word-length** | **sequencer** | **label** | **function** | **test** | **organelle** | **power-pin#** | etc.}
library	<definition> ::= {<macro> | <function> | <test> | <organelle>}
organelles.l	**layout**-<H>-**organelle**[2] <H> ::= { **inverter** | **nand** | **and** | **nor** | **xor** | **1+** | etc.}

The names, to a large degree, convey their function's purpose. For example, **layout-data-path** creates the L5 code for this portion of the chip. The library has many instantiations of the **definition**[3] **defstruct**. Examining this long structure generator [Figure 6.3] reveals that much of MacPitts syntax is data:

In particular, note that an <organelle>, <macro>, <function> or <test> are all **definition** cases. The library is a big list with parameters for specific

---

[2] **layout**-<H>-**organelle** is the L5 item that will be implemented in the circuit, whereas, **layout**-<X> is a function that lays out or creates items.

[3] Located in the MacPitts program defstructs.l.

creations of these data structures.[4] The functions that are implemented in the library are only constrained by the designer's imagination and a bit-slice regime.

The library is used by the **lookup-**<> function to correlate a function name found in a <MacPitts program> with an already created functional form. This can be shown in a simple example. Assume that the library is:

```
-> (setq library '((library)(constant t (nor))
                   (function 1+ ...)(test = ...)))<CR>
```

Then the equality test, =, can be found as follows:

```
-> (lookup-test '= library)<CR>
;; Find the " = "test definition form in the library.
     (test = ...)
```

In summary, MacPitts relies heavily on **defstruct**s. A <MacPitts program> is parsed and converted into a **defstruct** called an object. The five portions of this object are then converted into L5 by different programs. A particular set of layout functional units is included as a **defstruct** which contains information relating the unit to MacPitts' syntax in the library. A corresponding L5 layout of the unit is found in organelles.l. L5 in turn is a language composed of **defstruct**s and layout operators.

With the general idea in mind of how MacPitts coordinates its various parts to produce a chip, consideration is next given to modifying an organelle and implementing it functionally.

---

[4] In Section VI.B an example will be traced all the way from the layout to the test definition.

```
(defstruct definition
      library ()
      logo (text)
      word-length (value)
      eval (when lisp-form)
      power (pin#)
      ground (pin#)
      phia (pin#)
      phib (pin#)
      phic (pin#)
      register (name)
      flag (name)
      signal (name direction pin#)
      port (name direction pin#s)
      process (name)
      macro (name type lisp-form)
      constant (name value)
      organelle
       (name #control-lines #parameters
            #test-lines result? gen-form
            sim-form)
      function (name organelle-name types
            control-lines parameters
            interpret-form)
      test (name organelle-name types control-
            lines parameters test-line
            interpret-form)
      label (name process state)
      source (name)
      destination (name))
```

Figure 6.3 **definition defstruct**

B. AN EXAMPLE

The organelle used in this example was designed by Lieutenant Anthony Mullarky using Magic. The approach used was based on (Fox, 1983, p. 32). Like Fox, a modification was made to the equality test organelle to reduce its size and increase its speed. The organelle is implemented so that its result pulls down the output bus to Vss when the test fails. Two different cells are used: bit_0 and a bit_N. The zero bit organelle is a one bit equality checker tied to Vdd in order to precharge the output bus to +5 Volts. The Nth bit organelle is a one bit equality tester without a pullup. The appellation " == " is used to differentiate this equality test from MacPitts' " = ".

The first items needed are an **organelle-==-bit-0** and **organelle-bit-==-bit-n**. The organelles were made using Magic and output as CIF. The CIF was converter to Caesar format and then into L5 format. The two organelles are shown in Figures 6.4 and 6.5.

These two organelles are then incorporated into the standard MacPitts library. Organelles.l, the compiled portion of the library, is composed of a default set of MacPitts functional units in L5 format. Adders, decrementers, equality testers, etc., are all located in organelles.l. The L5 layouts are usually defsymbols and have a name of the form: **layout**-<X>-**organelle**. The two basic zero and Nth bit items were made into defsymbols without any arguments.

```
(defun layout-==-organelle (ratio bit)
;; Doesn't use the ratio input.
  (cond ((=0 bit)(organelle-==-bit-0))
        (t (organelle-==-bit-n))))
```

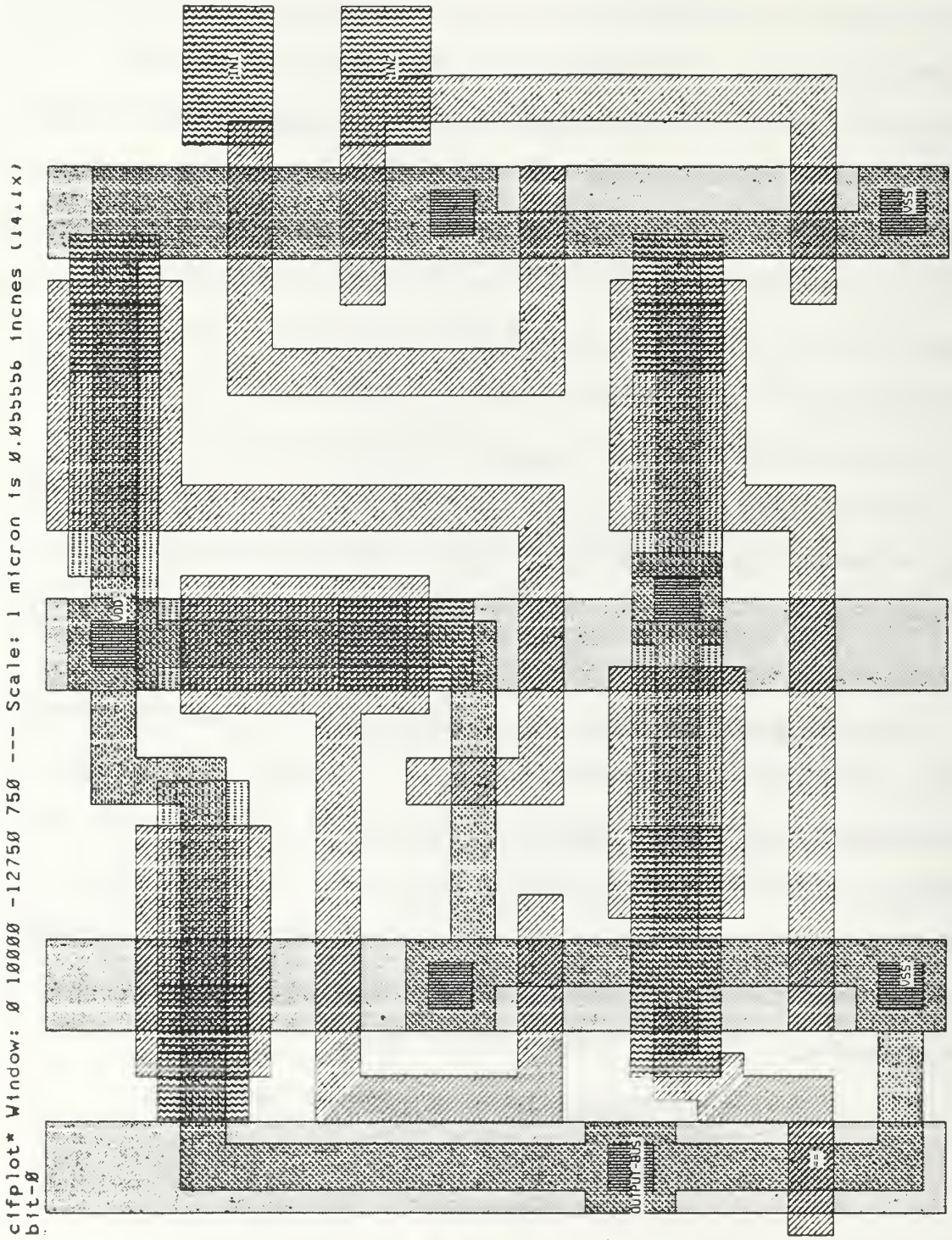

Figure 6.4 **(organelle-==-bit-0)**

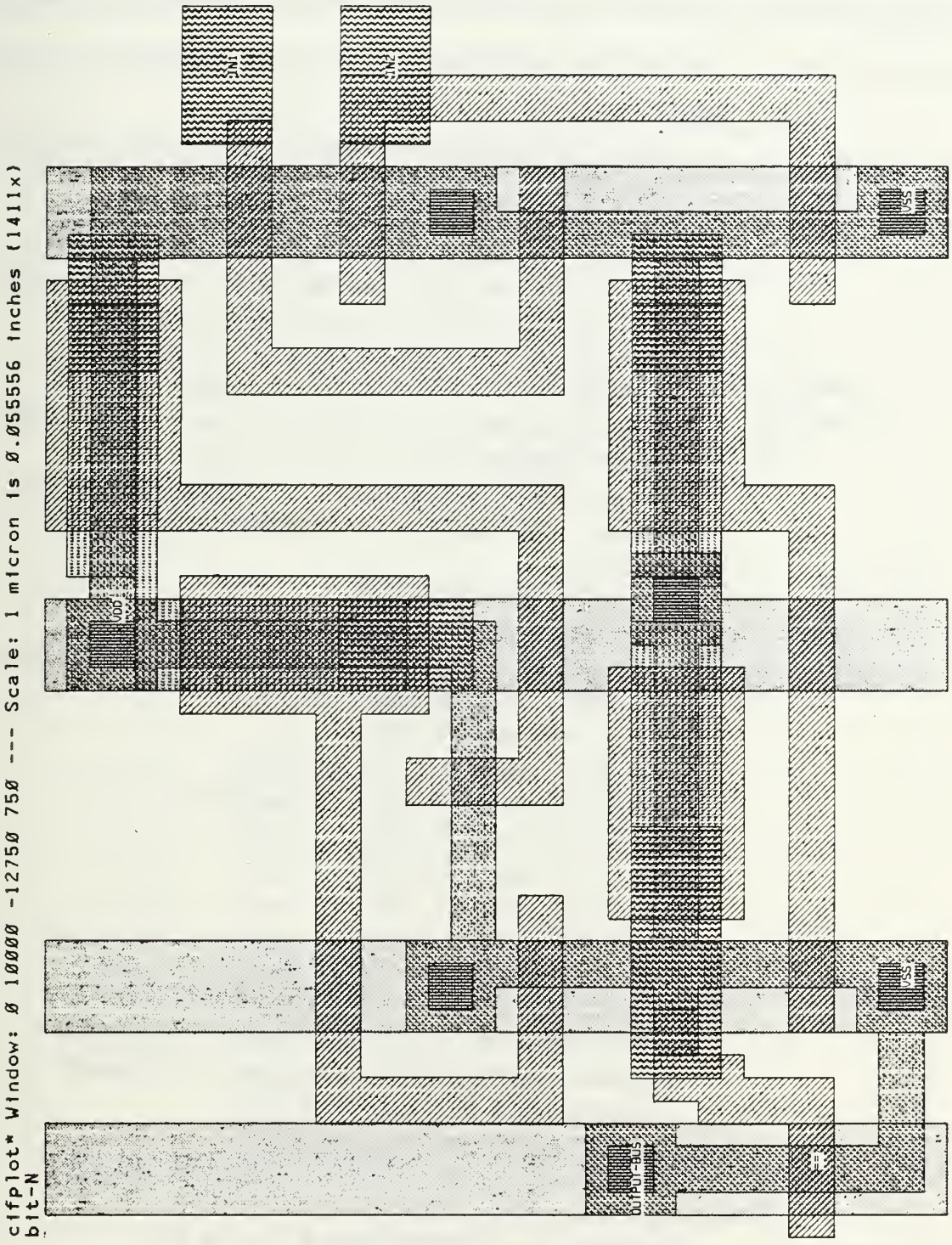

Figure 6.5 **(organelle-==-bit-n)**

Once this modification is made, then an organelle data structure is created and placed into the library [uncompiled]. Recall that an organelle is a specific case of a **definition defstruct**. [Refer to Appendix A]

```
(organelle == 0 2 1 no
      ;; <name> := ==, <#control-lines> := 0, <#parameters> := 2
      ;; <#test-lines> := 1, <result?> := no,
      ;; The <gen-form> field follows, notice that it is a
      ;; functional form. The <gen-form> is obtained by the
      ;; look-up-gen-form function. layout-gen-form creates it.
      ;; So, if this whole data structure is set to ==-test:
      ;; -> (setq ==-test '(organelle == 0 2 1 no (lambda . . .]
      ;; and the gen-form is extracted and named " create ",
      ;;->(setq create (organelle-definition-gen-form ==-test))
      ;; -> (apply create '(instantiate 0 nil nil nil))
      ;; Will create an instantiation of the organelle's bit zero.
       (lambda (info bit word-length drive ratio)
       (cond
        ((eq info 'instantiate)
         (first-quadrant
            (mirrorx (layout-==-organelle ratio bit))))
        ((eq info 'length) 58)
        ((eq info 'width) 48)
        ((eq info 'inputs) '(26 31))
        ((eq info 'output-type) '(ratio))
        ((eq info 'vdd) '(43))
        ((eq info 'gnd) '(8 28))
        ((eq info 'daisy) '(51))
        ((eq info 'test) '(51))
        ((eq info 'conductivity)
        (cond
         ((equal ratio '(4 4)) (quotient 1 0.7759))
         ((equal ratio '(8 8)) (quotient 1 0.7604))
         (t (quotient 1 0.7529))))
        ((eq info '#transistors) '(9 5))
         (t ())))
      ;; The <sim-form> is used by the simulator.
       (lambda (c a b)(list (cond ((== a b)
            '(1)) (t '(0))) ())))
```

Since " == " is a <test> operator, a test form is created to give the organelle functionality. This form is placed in the library along with the organelle data structure. The code for <test> " == " is:

```
(test == == (integer integer) ()
  ;; <name> := ==, <organelle> := ==,
  ;; <types> := (integer integer), <control-lines> := nil
  ;; <parameters> := ((position 1)(position 2))
  ;; <test-line> := (physical 1)
  ((position 1) (position 2)) (physical 1)
  ;; <interpret-form> follows:
  (lambda (form word-length x y)
      (cond ((or (eq x 'undefined-integer)
                 (eq y 'undefined-integer))
                 'undefined-boolean)
            ((= x y) 't)
            (t 'f))))
```

A MacPitts program is now run to check this new operator [Figure 6.6].

```
      (program five== 4
;; Example of a MACPITTS algorithm to test a 4-bit
;; integer's equality with the number five.
;; <filename> := five==.mac
(def 11 power)
(def 1 ground)
(def 2 phia)
(def 3 phib)
(def 4 phic)
(def in port input (5 6 7 8))
(def out signal output 9)
;; A reset pin is needed to initialize the chip.
(def reset signal input 10)
 (process equality 0
 first
 (cond
     ;; If " in " is " == " to 5 then set " out " to t.
     ((== in 5)(setq out t)(go first))
     ;; Otherwise, test " in " again.
     (t (go first)) ) ) )
```

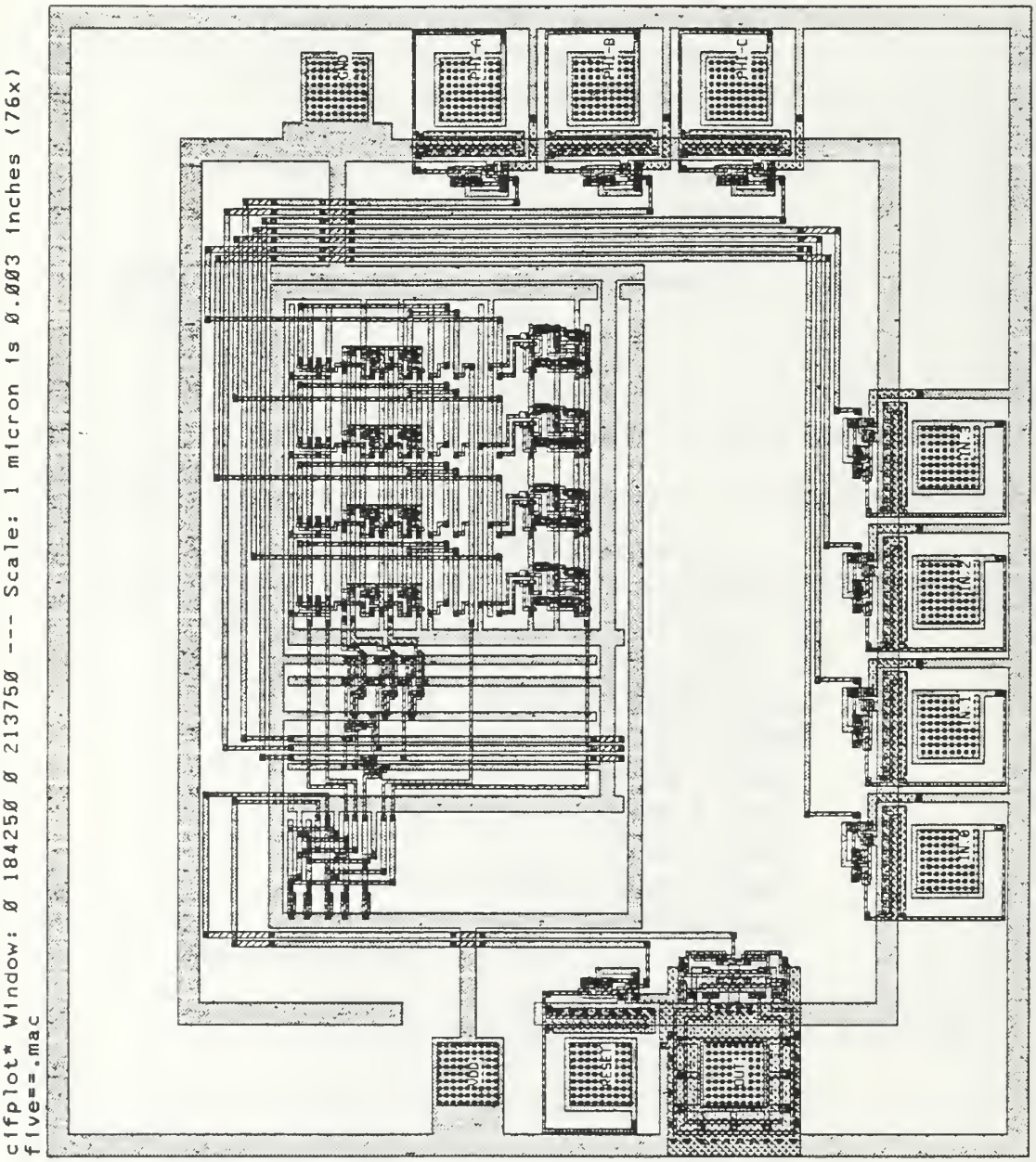

Figure 6.6 **five==.mac**

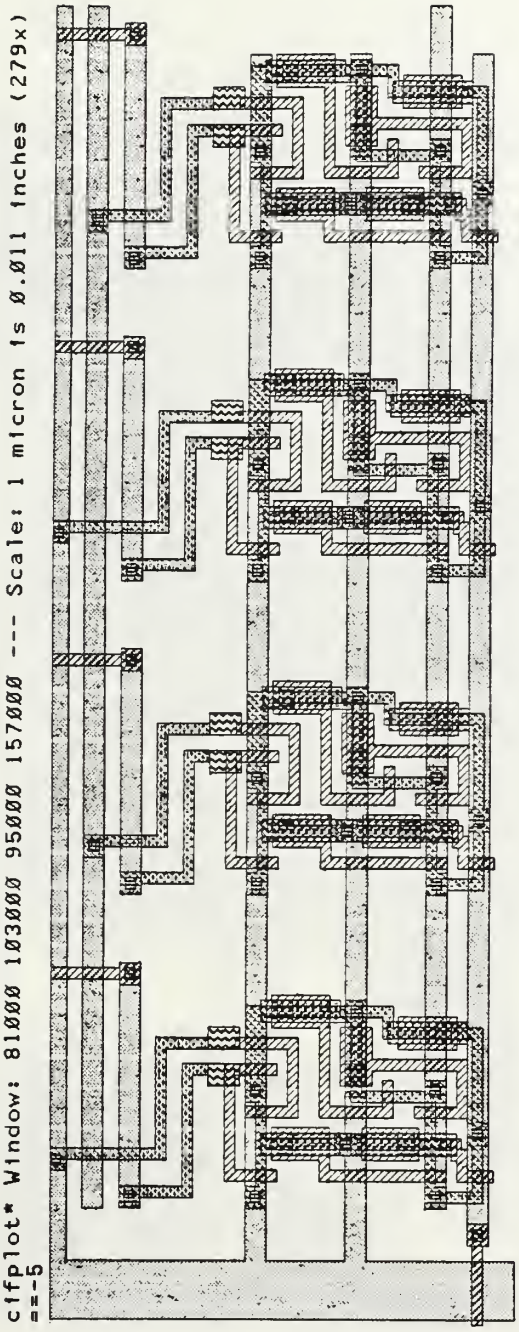

Figure 6.7 Closeup of == Organelle in **five==.mac**

The program was run with " = " and is shown below for comparison with Figure 6.6.

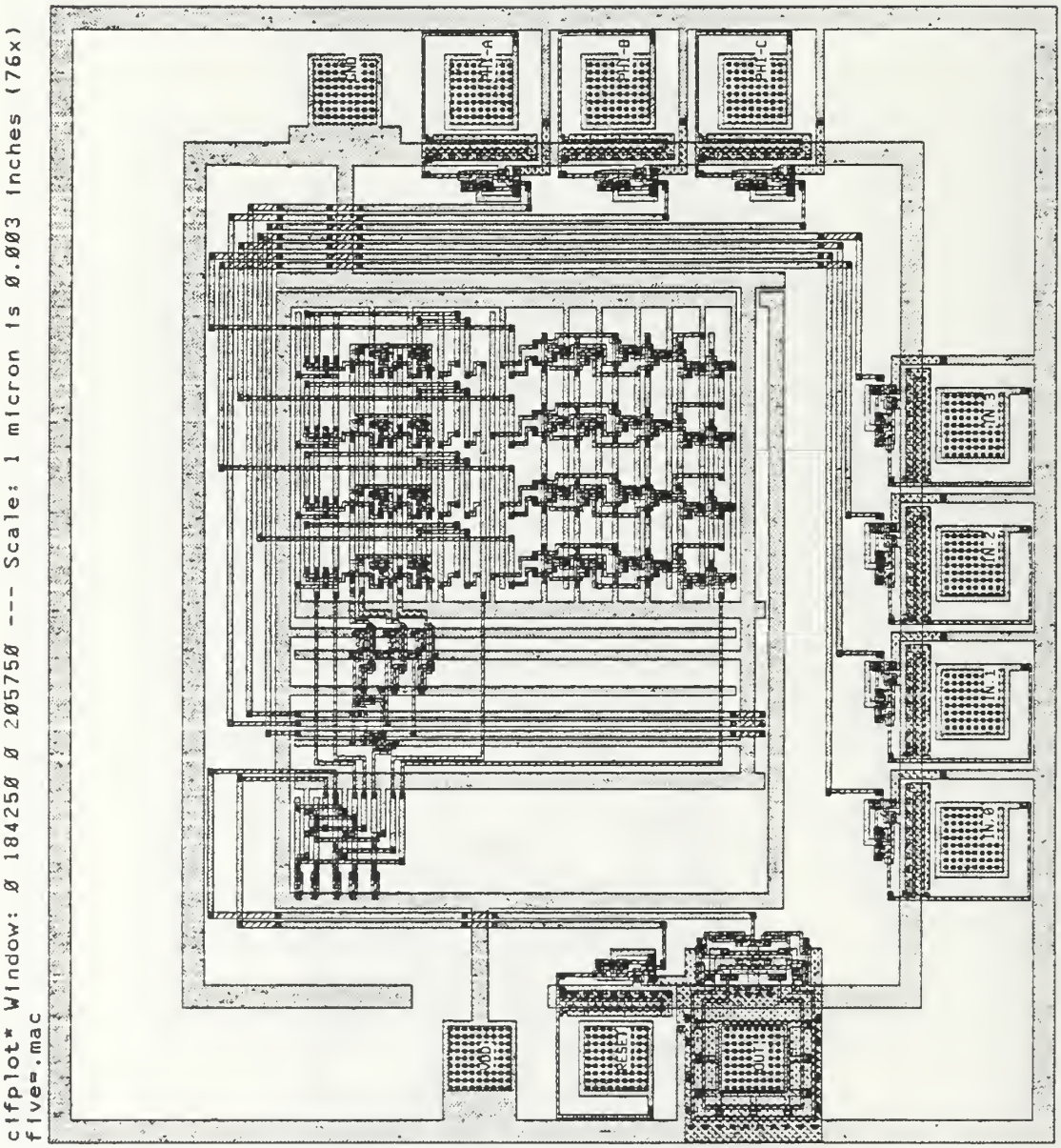

Figure 6.8 **five=.mac**

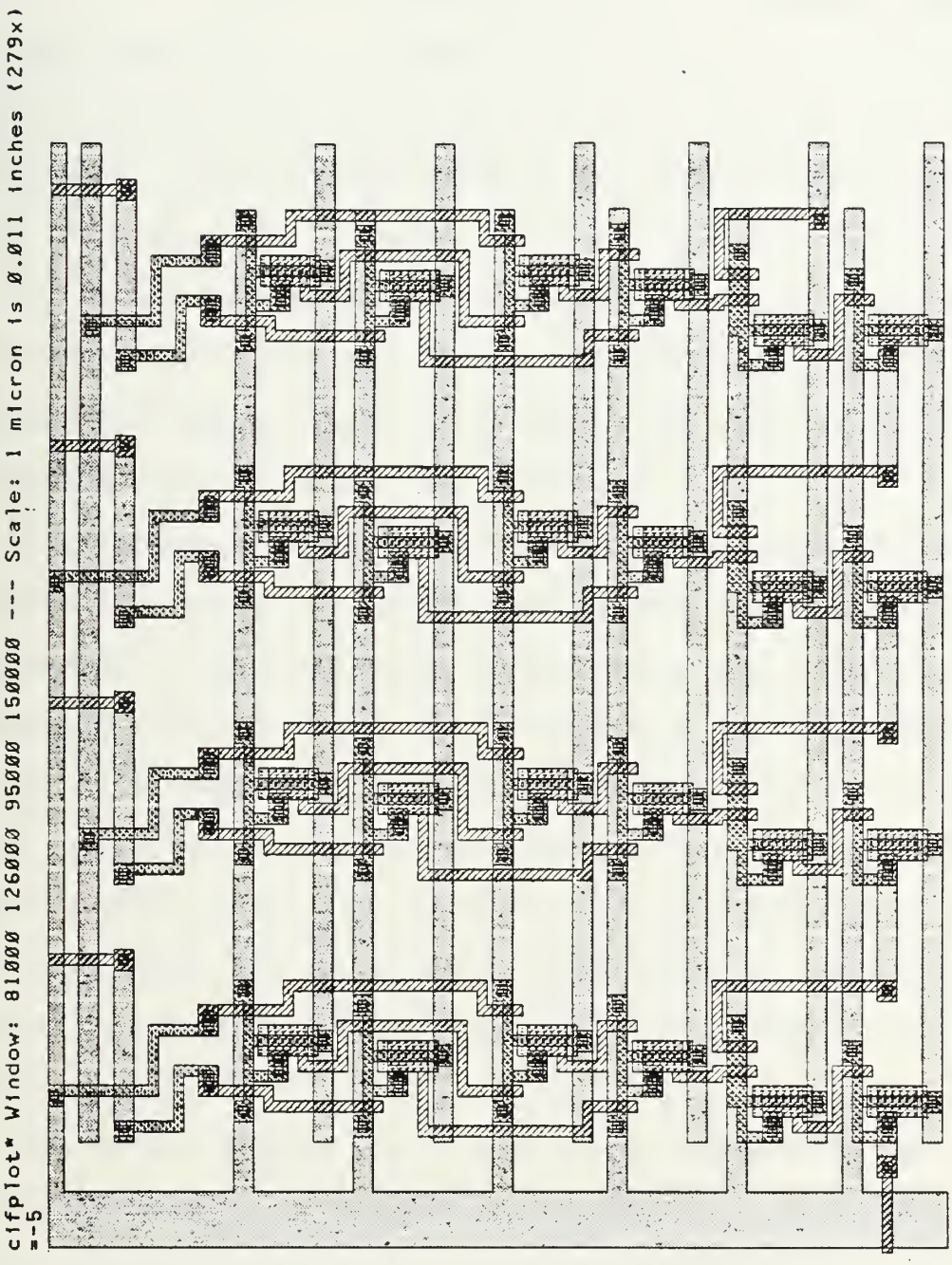

Figure 6.9 Closeup of = Organelle **five=.mac**

In summary, a MacPitts' functional unit was modified and the changes made at different levels were examined. The new " == " unit was significantly smaller that the standard equality organelle.

VII. CONCLUSIONS

This thesis' goal was to examine L5 and show how it is used in two silicon compilers: LBS and MacPitts. The thesis showed that L5 is an extension of LISP specifically aimed at VLSI synthesis. LISP's ability to treat functions as data to create new operators was found to be the basis for the versatile **defstruct** data structure generator. The main result of the thesis is the incorporation into one document of enough information to enable a VLSI designer to automate portions of the layout process or change existing MacPitts functions to meet other needs.

For example, an examination of L5's layout primitives and data structures showed its compatibility with Caesar and CIF. However, since the graphical editor now being used at the Naval Postgraduate School is Magic, a method for using this format with L5 is needed. The suggested approach is to use the structure of L5 programs that convert Caesar into L5 and vice versa; and instead, make the conversion directly from CIF to L5. This would make available a larger pool of circuits which have been converted to CIF for incorporation into the compilers. Additionally, it would buffer the system from other changes in graphical editor file format since CIF is a widely used format.

Many possible changes to MacPitts have been recommended by Carlson, Froede and Larrabee; among these suggestions, is to allow pin locations on all four sides of the chip. In light of what has been presented in this thesis this would be a fairly straightforward alteration.

The location of the bonding pads is controlled by the **layout-pins** function in the frame.l section of MacPitts. This program creates a frame that determines the architectural implementation of a MacPitts program. It places the different structural units into a Vdd/Vss skeleton and interconnects them. The input to its top level function, **layout-object**, accepts the object code generated by **get-object** in prepass.l. The main **layout-object** function in turn utilizes a **layout-pins** function [also found in frame.l]. A program that would place the pins for the user [at present they need to be specified] could be written. Other work could incorporate different routing schemes into frame.l.

From this point onward, changes are only limited by the user's inventiveness.

APPENDIX A: MISCELLANEOUS TOPICS

A. LAYOUT ERRORS

There are two types of errors associated with layouts created from Macpitts: either the code has been improperly written, or the output CIF is being plotted at the wrong scale.

An example of improper code is the erroneous specification of Vdd, Vss and output locations when a new organelle is created. In Section VI.B a new organelle, " == ", was input into MacPitts. If different parameters [shown in Table A.1] had been specified in the organelle structure, then the results of Figure A.1 would result. Notice that the interconnecting lines have all shifted to the right 3 λ units.

TABLE A.1

ORGANELLE SPECIFICATION COMPARISON

Correct	Erroneous
((eq info 'length) 58)	**((eq info 'length) 52)**
((eq info 'width) 40)	**((eq info 'width) 40)**
((eq info 'inputs) '(26 31))	**((eq info 'inputs) '(26 31))**
((eq info 'vdd) '(43))	**((eq info 'vdd) '(40))**
((eq info 'gnd) '(8 28))	**((eq info 'gnd) '(5 25))**
((eq info 'daisy) '(51))	**((eq info 'daisy) '(48))**
((eq info 'test) '(51))	**((eq info 'test) '(48))**

Additionally, the organelle length was specified to be several λ units longer than the layout actually is, to prevent the output line (when routed to the Weinberger array) from shorting with the clock lines [See Figure A.2].

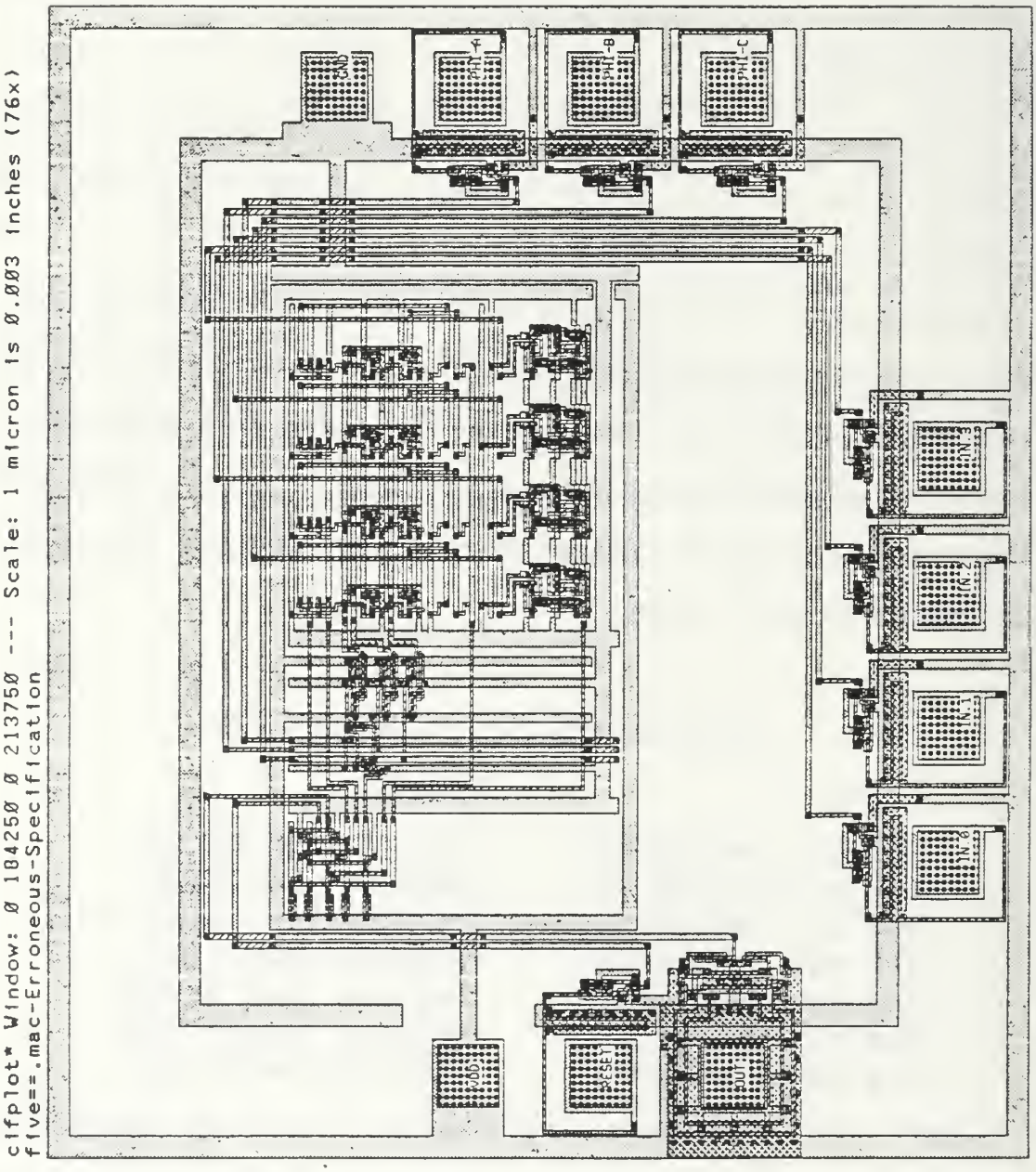

Figure A.1 **five==.mac** With an Incorrect Organelle Specification

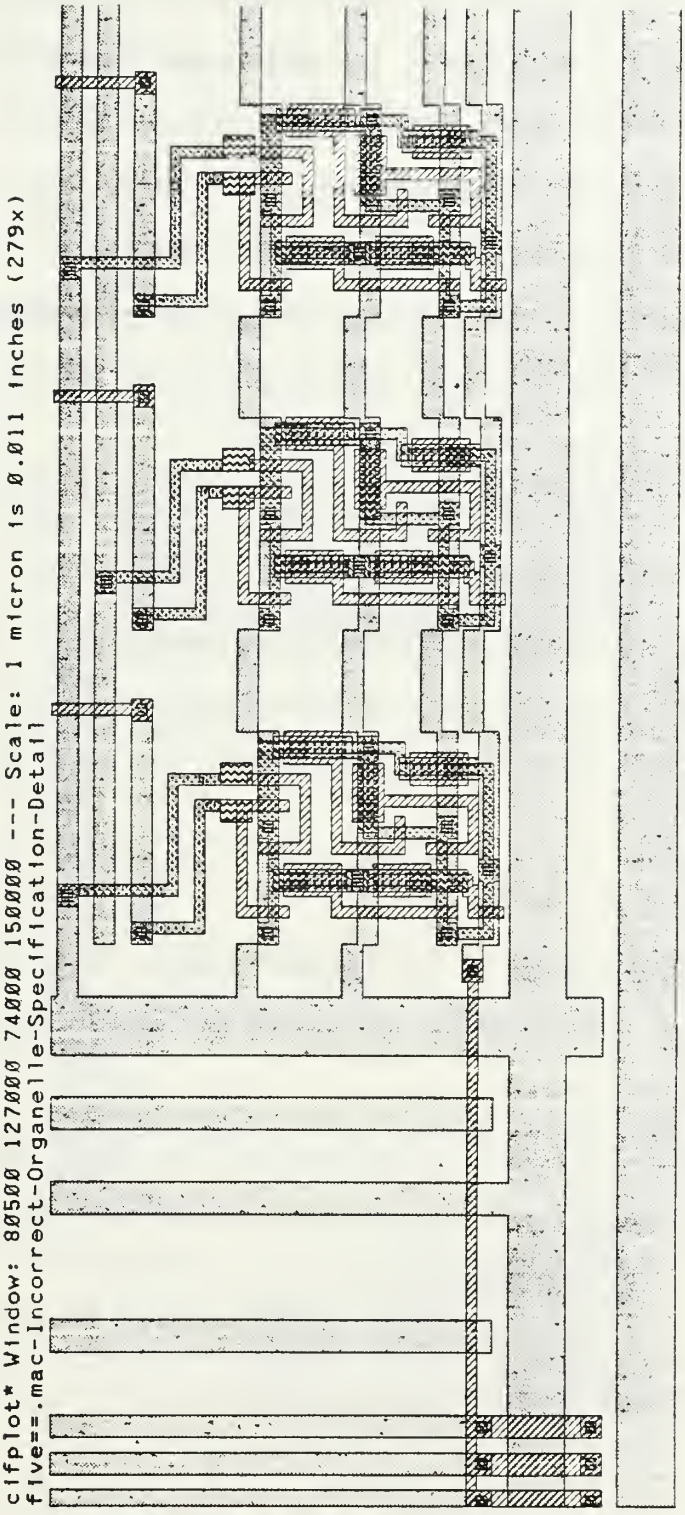

Figure A.2 **five==.mac** Organelle Detail

The other error type which is encountered is due to improper use of CIF plotting routines. For example, if the **minimum-feature-size** has been set to 250 centi-μ/λ and plotted using cifplot (which defaults to 200 centi-μ/λ), then plot misalignments will result. It should also be remembered that when **on-disk** storage is selected all items which are **defsymbols** are being created as CIF at the current **minimum-feature-size**. If the user should change the scale size in the middle of creating items, then the CIF files will be incompatible.

Compare Figures A.3 and A.4. In the first figure the pins from five=.mac were laid out without any data-path or controller. The connections from the clock lines can clearly be seen. This item was created using 200 centi-μ/λ and then plotted using cifplot at this same scale. Figure A.4, on the other hand was plotted using 250 centi-μ/λ.

B. EXPERIMENTING IN MACPITTS

A quick look at the use of the **layout-object** function [in the frame.l program] to modify pin locations is used as an introduction to future topics of investigation.

It was seen in Section VI.C that the MacPitts environment is accessible because the top-level function is set up to do this as follows:

```
% macpitts
usage: macpitts <filename> [<options>]

[Return to top level]
->
```

Figure A.3 **five=.mac** Pins With Correct CIF Scale

Figure A.4 **five=.mac** Pins With Erroneous CIF Scale

The user now has all of MacPitts available with the exception of the organelles and library.

Instead of running MacPitts every time a change is made, the intermediate object code will be modified. Object code is generated when five==.mac is run in the following manner:

```
% macpitts obj cif noint noopt-c noopt-d
nostat five==.mac & > trash
*Create an object and CIF file. Redirect comments
*to a trash file.
```

Now, assuming that the macpitts environment has been invoked as shown above [**% macpitts**], two local files that contain commands to change UNIX® directories and plot L5 items are loaded.

```
-> (include edit.l)
  [load edit.l]
t

-> (include plot.l)
[load plot.l]
t

-> (minimum-feature-size! 200)
200
```

The object file that was generated by MacPitts, five==.obj, is altered by setting its data-path and controller to **nil**. After editing is complete, the new file is called pin-test.obj and is shown below:

```
-> (exec cat pin-test.obj)
;; The first portion of the object is definitions
(((source reset)
(register sequencer-equality-state)
(source sequencer-equality-state)
(destination sequencer-equality-state)
```

```
(port sequencer-equality-next-state internal
                                                nil)
(source sequencer-equality-next-state)
(destination sequencer-equality-next-state)
(label first equality 0)
(destination out)
(source first)
(source in)
(logo five=)
(word-length 4)
 (power 11)
 (ground 1)
 (phia 2)
 (phib 3)
 (phic 4)
 (port in input (5 6 7 8))
 (signal out output 9)
 (signal reset input 10)
 (process equality))
;; No flags were used in five==.mac
 nil
;; The data-path portion of five==.obj has been set to nil.
 nil
;; The control segment has also been set to nil.
 nil
;; The pin section of the object remains intact.
 ((4 (phic))
  (3 (phib))
  (2 (phia))
  (1 (ground))
  (11 (power))
  (9 (output8 out (signal-output out)))
  (5 (input (in 3) (port-input in 3)))
  (6 (input (in 2) (port-input in 2)))
  (7 (input (in 1) (port-input in 1)))
  (8 (input (in 0) (port-input in 0)))
  (10 (input reset (signal-input reset)))))
```

This object is turned into an L5 item by the **layout-object** function found in the frame.l program.

```
-> (thesis-plot (layout-object
    (read (infile 'pin-test.obj))) 'pin-test.obj t)
;; The standard statistics are output, except the control
;;  unit and data path are empty.
Statistic - Control has 0 columns
Statistic - Circuit has 78 transistors
Statistic - Control has 0 tracks
Statistic - Power consumption is 0.034114
      Watts
Statistic - Data-path internal bus uses 0 tracks
Statistic - Dimensions are 1.640000 mm by
      1.718000 mm
;; The rest of the output is related to the plotting
;; function.
. . .
-> Window: 0 164000 0 171800
Scale: 1 micron is 0.002805 inches (71x)
The plot will be 0.38 feet
```

This plot is shown in Figure A.3. The object file is modified again to place the pins in different locations:

```
-> (exec cat pin-test-2.obj)
;; A random pin ordering was chosen.
(((source reset)
  (register sequencer-equality-state)
  (source sequencer-equality-state)
  (destination sequencer-equality-state)
  (port sequencer-equality-next-state internal
                                          nil)
  (source sequencer-equality-next-state)
  (destination sequencer-equality-next-state)
  (label first equality 0)
  (destination out)
  (source first)
  (source in)
  (logo five=)
  (word-length 4)
  (power 11)
  (ground 1)
  (phia 2)
```

```
(phib 5)
(phic 10)
(port in input (3 4 7 9))
(signal out output 8)
(signal reset input 6)
(process equality))
nil
nil
nil
((10 (phic))
 (5 (phib))
 (2 (phia))
 (1 (ground))
 (11 (power))
 (8 (output8 out (signal-output out)))
 (9 (input (in 3) (port-input in 3)))
 (7 (input (in 2) (port-input in 2)))
 (4 (input (in 1) (port-input in 1)))
 (3 (input (in 0) (port-input in 0)))
 (6 (input reset (signal-input reset)))))
```

The results of this change in pin locations is shown in Figure A.5. The intent is not that the user make tedious changes to object code to optimize pin layout; but rather, that this process be automated or modified for other packaging schemes.

In summary, this appendix described some observed errors that the user should avoid and introduced an area for future work.

Figure A.5 **five=.mac** Modified Pins.

LIST OF REFERENCES

Aho, A.V., Sethi, R., and Ullman, J.D., Compilers: Principles, Techniques and Tools, Addison-Wesley, 1986.

Ayres, R. F., VLSI: Silicon Compilation and the Art of Automatic Microchip Design, Prentice-Hall, 1983.

Brooks, R. A., Programming in Common LISP, John Wiley & Sons, 1985.

Carlson, D. J., Application of a Silicon Compiler to VLSI Design of Digital Pipelined Multipliers, M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1984.

Charniak, E. and McDermott, D., Introduction to Artificial Intelligence, Addison-Wesley, 1985.

Crouch, K. W., L5 User's Guide, Massachusetts Institute of Technology Lincoln Laboratories Project Report RVLSI-5, 7 March 1984.

Defense Advanced Research Project Agency, STRATEGIC COMPUTING, New-Generation Computing Technology: A Strategic Plan for its Development and Application to Critical Problems in Defense, 1983.

Dreyfus, H.L., "From Micro-Worlds to Knowledge Representation: AI at an Impasse", in Haugeland, Mind Design, 1979, pp. 161-204; and, Dreyfus, H.L., What Computers Can't Do, Harper and Row, 1979.

Flew, A., A Dictionary of Philosophy, 2nd ed., St. Martin's Press, 1979.

Foderado, J. K., Sklower, K. L. and Layer, K., The Franz Lisp Manual, University of California at Berkeley, June 1983.

Froede, A. O.., Silicon Compiler Design of Combinational and Pipeline Adder Integrated Circuits, M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1985.

Gray, P.M.D., Logic, Algebra and Databases, Ellis Horwood Limited, 1984.

Hasemer, T., Looking at LISP, Addison-Wesley, 1984.

Haugeland, J., Artificial Intelligence: The Very Idea, MIT Press, 1985.

Haugeland., J., editor, Mind Design: Philosophy-Psychology-Artificial Intelligence, MIT Press, 3rd ed., 1985.

Hofstadter, D.R, Metamagical Themas, Basic Books, 1985.

Hon, R.W., and Sequin C.H., A Guide to LSI Implementation, Xerox Palo Alto Research Center, 1980.

International Business Machines (IBM) Technical Newsletter GC26-3847-5, APL Language, 6th ed., IBM Corporation, 1983.

Larrabee, R. C.., VLSI Design with the MacPitts Silicon Compiler, M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1985.

MacLennan, B. J., Principles of Programming Languages: Design Evaluation and Implementation, Holt, Rinehart and Winston, 1983.

Mead, C. and Conway, L., 2d ed., Introduction to VLSI Systems, Addison-Wesley, 1980.

Ousterhout, J.K., Editing VLSI Circuits with Caesar, Computer Science Division, Electrical and Computer Sciences, University of California, 1985.

Rosenberg, J.M., Dictionary of Computers, Data Processing and Telecommunications, John Wiley and Sons, 1984.

Scott, W.S., Hamachi, G., Mayo, R.N. and Ousterhout, editors, J.K., 1985 VLSI Tools: More Works by the Original Artists, Computer Science Division EECS Department, University of California at Berkeley, 1985.

Scott, W.S., Hamachi, G., Mayo, R.N. and Ousterhout, editors, J.K., 1986 VLSI Tools: Still More Works by the Original Artists, Computer Science Division EECS Department, University of California at Berkeley, 1986.

Siskind, J. M., Southard, J. R. and Crouch, K. W., "Generating Custom High Performance VLSI Designs from Succint Algorithmic Descriptions", MIT Conference on Advanced Research in VLSI, 1982.

Southard, J. R., An Introduction to MacPitts, Massachusetts Institute of Technology Lincoln Laboratories Project Report RVLSI-3, 10 February 1983.

Southard, J. R., Domic, A., Crouch, K. W., LBS- Lincoln Boolean Synthesizer, ESD-TR-82-087 Lincoln Laboratory Technical Report 622, 1982.

Southard, J. R., Domic, A., Crouch, K. W., "A Report on the LIncoln Boolean Synthesizer (LBS)", IEEE International Conference on Computer Aided Design (ICCAD 83), 1983.

Southard, J. R., "Macpitts: An Approach to Silicon Compilation", Computer, Volume 16, Number 12, 1983.

Wilensky, R., Lispcraft, W. W. Norton & Co., 1984.

Winston, P. E. and Horn, B. K., LISP, 2d ed., Addison-Wesley, 1984.

BIBLIOGRAPHY

ACM IEEE Design Automation Conference Proceedings, 19th, IEEE Press, 1982.

Agre, P. E., "Designing A High-Level Silicon Compiler", Proceedings IEEE International Conference on Computer Design: VLSI in Computers (ICCD 83), November 1983.

Allen, G.H., Denyer, P.B., Renshaw, D., "A Bit Serial Linear Array DFT", Proceedings of the IEEE Conference on Acoustics, Speech and Signal Processing (ICASSP 84), March 1984.

Allerton, D.J., Batt, D.A., and Currie, A.J., "A VLSI Design Language Incorporating Self-Timed Concurrent Processes", IEE European Conference on Electronic Design Automation (EDA 84), March 1984.

Anceau, F., "CAPRI: A Design Methodology and a Silicon Compiler for VLSI Circuits Specified by Algorithms", in Bryant, 1983, pp. 15-31.

Anceau, F., Aas, E.J. (editors), 'VLSI 83: Proceedings of the IFIP TC WG 10.5 International Conference on Very Large Scale Integration, North-Holland, 1983.

Anceau, F., and Schoellkopf, J.P., "CAPRI: A Silicon Compiler for VLSI Circuits Specified by Algorithms", in Randell, 1983, pp. 149-54.

Ayres, R., "Silicon Compilation-A Hierarchical Use of PLA's", Proceedings of the 16th Design Automation Conference, 1979.

Bairstow, J.N., "Chip Design Made Easy", High Technology, Volume 5, Number 6, June 1985.

Baker, A., "Selecting a Silicon Compiler", VLSI Systems Design, Volume VII, Number 5, May 1986.

Baker, S., "Silicon Compilers Puts Systems Houses in VLSI Business", Electronic Engineering Times, No. 300, 8 October 1984.

Baker, S., "Silicon Bits", Electronic Engineering Times, Number 303, 29 October 1984.

Baudet, G.M., Cutler, M., Davio, M., Peskin, A.M., and Rammig, F.J., "The Relationship Between HDLs and Programming Languages", in Workshop Report: VLSI and Software Engineering Workshop, 1984, pp. 64-69.

Beedie, M., "Silicon Compiler Takes Process Characteristic Into Consideration", Electronic Design, Volume 32, Number 14, 12 July 1984.

Beedie, M., "On the European Front, Silicon Compilers Focus on Design Languages", Electronic Design, Volume 32, Number 22, 31 October 1984.

Benschop, N. F., "Layout Compiler for Variable Array-Multipliers", Proceedings of the Custom Integrated Circuits Conference, May1983.

Beresford, R., "Advances in Customization Free VLSI System Designers", Electronics, 10 February1983.

Bergmann, N., "A Case Study of the F.I.R.S.T. Silicon Compiler", in Bryant, 1983, pp. 413-430.

Bergmann, N., Software Support for F.I.R.S.T., Edingurgh University Internal Report, June1982.

Bertolazzi, P., and Luccio, F. (editors), VLSI: Algorithms and Architectures, North-Holland, 1985.

Bilgory, A., Gajski, D.D., "An Algorithm for Efficient Layouts of Parallel Suffix Solutions", Proceedings of the 1981 International Conference on Parallel Processing, August 1981, IEEE Computer Society, 1981.

Birkner, J., "Macro's for Programmable Logic", WESCON/82 Conference Record 21-2/1-4, Electronic Conventions, 1982.

Blackman, T., Fox, J., and Rosebrugh, C., "The SILC Silicon Compiler: Language and Features", Proceedings 22nd ACM/IEEE Design Automation Conference, June 1985.

Bloom, M., "Silicon Compilation: The Fast Track to Masks", Electronic Engineering Times, Number 294, 13 August 1984.

Bond, J., "Circuit Density and Speed Boost Tomorrow's Hardware", Computer Design, Volume 23, Number 10, September 1984.

Bond, J., "Future Hardware", Computer Design, Volume 23, Number 10, September 1984.

Brayton, R.K., Chen, C.L., McMullen, C.T., Otten, R.H., and Yamour, Y.J., "Automated Implementation of Switching Functions as Dynamic CMOS Circuits", Proceedings of the 1984 Custom Integrated Circuits Conference, May 1984.

Brayton, R.K., Hachtel, G.D., McMullen, C.T., and Sangiovanni-Vincentelli, A.L., Logic Minimization Algorithms for VLSI Synthesis, Kluwer Academic Publishers, 1984.

Brown, C. "Silicon Compilers: How Some Labs are Trying to Implement Them", Electronic Engineering Times, No. 300, 8 October 1984.

Bryant, R., editor, Third Caltech Conference on Very Large Scale Integration, Computer Science Press, 1983.

Buric, M.R., Christensen, C., and Matheson, T.G., "The Plex Project: VLSI Lay outs of Microcomputers Generated by a Computer Program", IEEE International Conference on Computer-Aided Design (ICCAD-83), September1983.

Bursky, D., "Circuit Compiler Cuts Chip Area 25 Percent to 40 Percent Over Standard Cells", Electronic Design, Volume 32, Number 18, 6 September 1984.

Carnegie-Mellon University Departments of Computer Science and Electrical Engineering Technical Report, by Barbacci, M. R., Barnes, G., Cattell, R. and Siewiorek, D., The ISPS Computer Description Language, 16 August 1979.

Carnegie-Mellon University Departments of Computer Science and Electrical Engineering Technical Report, by Barbacci, M. R., et. al. , The Symbolic

Manipulation of Computer Descriptions: The ISPS Computer Description Language, March 1978.

Capello, P. R., et. al. editors, VLSI Signal Processing, IEEE Press, 1984.

Carter, T.M., Davis, A., Hayes, A.B., Lindstrom, G., Klass, D., Maloney, M.P., Nelson, B.E., Organick, E.I., and Smith, K.F., "Transforming an ADA Program Unit to Silicon and Testing it in an ADA Environment", Digest of Papers COMPCON Spring 84. Twenty-Eighth IEEE Computer Society International Conference, March 1984.

Cham, K.M., Oh, S.-H., Chin, D., and Moll J.L., Computer Aided Design and VLSI Device Development, Kluwer Academic Publishers, 1985.

Cheng, E.K., "The Design of an Ethernet Data Link Controller Chip", Digest of Papers Spring COMPCON 83. Intellectual Leverage for the Information Society, March 1983.

Clary, J.B., Denyer, P.B. (editors), "The Impact of VLSI on Digital System Design", Systems on Silicon. First IEE International Specialist Seminar 2, Peter Peregrinus, 1984.

Cline, K., Cutler, M., Kesselman, C., and York, G., "Automated Attribute Optimization for VLSI Systems", 1984 Conference Proceedings- 3rd Annual Intenational Phoenix Conference on Computers and Communications, IEEE Press, 1984.

Collet, R., "Silicon Compilation: A Revolution in VLSI Design", Digital Design, Volume 14, Number 8, August 1984.

Conference Record-16th Asilomar Conference on Circuits Systems & Computers, IEEE Press, 1982.

Cory, W.E., and van Cleemput, W.M., "Developments in Verification of Design Correctness", Proceedings 17th Design Automation Conference, 1980.

Curtis, W., "Silicon Compilation Speeds Design of Complex Chips", Computer Design, March 1985.

Curtis, W., "Designing the Micro-VAX Using Silicon Compilation", COMPCON '85 Computer Conference, IEEE Computer Society, 1985.

Cuykendall, R., Domic, A., Joyner, W.H., Johnson, S.C., Kelem, S., McBride, D., Mostow, J., Savage, J.E., and Saucier, G., "Design Synthesis and Measurement", in Workshop Report: VLSI and Software Engineering Workshop, 1984, pp. 6-9; and, Journal of Systems and Software, Volume 4, Number 1, April 1984.

Darringer, J. A. and Joyner, W. H., Jr., A "New Look at Logic Synthesis", Proceedings of the 17th Design Automation Conference, 1980.

Darringer, J. A. et. al., "Logic Synthesis Through Local Transformations", IBM Journal of Research and Development, Volume 25, Number 4, 1981.

Dasgupta, S., "Computer Design and Description Languages", Advances in Computers, Volume 21, Academic Press, 1982.

DeMan, H. et. al., "Custom Design of Hardware Digital Filters on I.C's", Proceedings of the 1982 Custom Integrated Circuits Conference, 1982.

DeMan, H., Reynders. L, Bartholomeus, M., Cornelissen, J., "PLASCO: A Silicon Compiler for NMOS and CMOS PLA's", in Anceau, 1983, pp 171-81.

Denyer, P. B., Renshaw, D., and Bergmabb, N., "A Silicon Compiler for VLSI Signal Processors", Proceeding of the 8th European Solid-State Circuits Conference (ESSCIRC 82), 1982.

Denyer, P.B., and Renshaw, D., "Case Studies in VLSI Signal Processing Using a Silicon Compiler", Proceedings ICASSP, April 1983.

Denyer, P.B., Murray, A.F., and Renshaw, D., "FIRST: Prospect and Retrospect", in Capello, 1984, pp. 252-264.

Denyer, P. and Renshaw, D., VLSI Signal Processing, A Bit-Serial Approach, Addison-Wesley, 1985.

Digest of Papers- COMPCON Spring 82: High Technology in the Information Industry, IEEE Computer Society, 1982.

Digest of Technical Papers- IEEE International Conference on Computer-Aided Design, ICCAD-83, IEEE Press, 1983.

Dorsch, J., "Station Vendors Join Silicon Compilation Action", Electronic News, Volume 30, No. 1524, 19 November 1984.

Dreyfus, H.L., and Harrison, H., Husserl, Intentionality and Cognitive Science, The MIT Press, 1984.

Elias, N. and Wetzel, A., "The IC Module Compiler: a VLSI Systems Design Aid", International Circuits and Computers Conference, 1982.

Evans, W.H., and Allen, J., "MOS Implementations of TTL Architectures: A Case Study", Proceedings ICASSP, 1982.

Feldman, J.A. and Beauchemin, E.J., "A Custom IC for Automatic Gain Control in LPC Vocoders", Proceedings- ICASSP 83, IEEE, April 1983.

Feldman, S.I, "The Circuit Design Language Xi", Proceedings IEEE International Conference on Computer Design (ICCD-83), October 1983.

Fields, S.W., "Silicon Compiler Cuts Time to Market for DEC's MicroVAX I", Electronics, Volume 56, Number 21, 20 October 1983.

Floyd, R.W., "The Compilation of Regular Expressions into Integrated Circuits", Proceedings of the 21st Annual Symposium of the Foundations of Computer Science, 1980.

Foster, M.J. and Kung, H. T., "Recognize Regular Languages with Programmable Building-Blocks", in Gray, 1981, pp. 75-84.

Fox, J.R., "Performance Prediction with the MacPitts Silicon Compiler (for VLSI Circuit Synthesis)", Proceedings of the 1984 Custom Integrated Circuits Conference, May 1984.

Franklin, W. R., "Software Engineering Lessons for VLSI Design Methodology", in Workshop Report: VLSI and Software Engineering Workshop, 1984, pp. 86-89.

Frantz, D., and Rammig, F.J., "The Impact of an Advanced CHDL in VLSI Design", Proceedings International Conference on Computer Design (ICCD-83), October 1983.

"Frustration Lead Alles and Rip into the Custom-IC Business", Electronics, Volume 57, Number 13, 28 June 1984.

Fujita, T., and Goto, S., "A Rule Based Routing System", ICCD-83.

Furlow, B., "Silicon Compilation Cuts Costs of Custom VLSI", Computer Design, Volume 23, Number 12, 15 October 1984.

Gajski, D.D., "The Structure of A Silicon Compiler", IEEE International Conference on Circuits and Computers 1982 (ICCC 82), IEEE Press,1982.

Gajski, D.D. and Kuhn, R.H.," Guest Editors' Introduction: New VLSI Tools", Computer, Volume 16, Number 12, 1983.

Gajski, D.D., "Silicon Compilers and Expert systems for VLSI", Proceedings '84 ACM IEEE 21st Design Automation Conference, June 1984.

Gazsi, L., "Explicit Formulas for Lattice Wave Digital Filters", IEEE Transactions on Circuits and Systems, Volume CAS-32, Number 1, January 1985.

Glasser, L. A. and Dobberpuhl, D. W., The Design and Analysis of VLSI Circuits, Addison-Wesley, 1985.

"Graphics Systems Chip Designed to Replace 80 TTL Components", Electronic Engineering Times, Number 284, 23 April 1984.

Gray, J.P., editor, VLSI 81. Very Large Scale Integration. Proceedings of the First International Conference on Very Large Scale Integration, Academic Press, 1981.

Gray, J.P., Buchanan, I. and Robertson, P.S., "Controlling VLSI Complexity Using a High-Level Language for Design Description", Proceedings IEEE International Conference on Computer Design (ICCD-83), October 1983.

Gray, J.P., Buchanan, I. and Robertson, P.S., "Designing Gate Arrays Using a Silicon Compiler", Proceedings of the 19th ACM IEEE Design Automation Conference, IEEE Press, 1982.

Gray, J.P., "Introduction to Silicon Compilation", Proceedings of the 16th Design Automation Conference, 1979.

Gray, J.P., Denyer, P.B. (editors), "Silicon Compilers (History and Analogy with Program Compilers)", Systems on Silicon. First IEE International Specialist Seminar 51, Peter Peregrinus, 1984.

Greene, J., and El Gamal, A., "Area and Delay Penalties in Wafer-Scale Arrays", in Bryant, 1983, pp. 165-184.

Gross, R.R., Silicon Compilers: A Critical Survey, Department of Computer Science, University of North Carolina at Chapel Hill Technical Report, 1983.

Guterl, F., "In Pursuit of the One-Month Chip", IEEE Spectrum, September 1984.

Hafer, L.J. and Parker, A.C.," Automated Synthesis of Digital Harware", IEEE Transactions of Computers, Volume C-31, Number 2, 1982.

Harris, M.A., "News Update", Electronics, January 1983.

Harstenstein, R.W., "Basics of Structured Design Methodologies: Data Path and Finite State Machines", in Jespers, 1980, pp. 73-108.

Hedges, T.S., Slater, K.H., Clow, G.W., and Whitney, T., "The SICLOPS Silicon Compiler", IEEE International Conference on Circuits and Computers (ICCC 82), October 1982.

Heidegger, Hofstadter, A. (trans.), The Basic Problems of Phenomenology, Indiana University Press, 1982.

Hennessy, J.,"SLIM: A Simulation and Implementation Language for VLSI Microcode", Lambda, Volume 2, 1981.

Hicks, P.J. (editor), Semi-Custom IC Design and VLSI, Peter Peregrinus, 1983.

Hilfinger, P., "Silage: A High Level-Language and Silicon Compiler for Digital Signal Processing", Proceedings CICC 1985, 1985.

Hirschhorn, S., and Davis, A.M., "The Revolution in VLSI Design: Parallels Between Software and VLSI Engineering",[in Conference Record-16th Asilomar Conference on Circuits Systems & Computers, 1982; and in Workshop Report: VLSI and Software Engineering, 1984, pp. 75-84.

Hodges, D.A., and Jackson, H.G., Analysis and Design of Digital Integrated Circuits, McGraw Hill, 1983.

Holland, J. H., Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence, The University of Michigan Press, 1975.

IEEE 1984 Workshop on the Engineering of VLSI and Software, IEEE Press, 1984.

Ishii, J., Sugiura, Y., and Sueshiro, Y., "A Gate Array CAD System and Future Tasks in the Field", Third Symposium on VLSI Technology, September 1983.

Jespers, P., Sequin, C., and van de Wiele, F., Design Methodologies for VLSI Circuits, Sitjthoff and Noordhoff, 1982.

Johannsen, D.L., "Bristle Blocks- A Silicon Compiler", Proceedings of the 16th Design Automation Conference, 1979.

Johannsen, D.L., Silicon Compilation, PhD. Dissertation, Technical Report 4530, California Institute of Technology, 1981.

Johnson, S.C., "Code Generation for Silicon", Proceedings 10th ACM Symposium on Principles of Programming Languages, 1983.

Johnson, S.C., "VLSI Circuit Design Reaches the Level of Architectural Description", Electronics, Volume 57, Number 9, 3 May 1984.

Johnson, S.C., and Mazor, S., "Silicon Compiler Lets System Makers Design Their Own VLSI Chips", Electronic Design, Volume 32, Number 20, 4 October 1984.

Joyner, W.H., "Compilation Techniques in Logic Synthesis", Workshop Report, VLSI and Software Engineering Workshop, Port Chexter, NY, October 1982, IEEE Computer Scociety Press, 1983.

Jones, W. T., A History of Western Philosophy, Harcourt Brace Jovanovich, Inc., 1970.

Kang, S., and van Cleemput, W.M., "Automatic PLA Synthesis from a DDL-P Description", Proceedings 18th Design Automation Conference, 1981.

Katz, R., Scacchi, W., and Subrahmanyam, P., "Development Environments for VLSI and Software Engineering", in Workshop Report: VLSI and Software Engineering Workshop, 1984, pp. 50-63.

Kejelkerud, E., Hedengran, U., and Karlsson, P., "Experiences in the Use of a Translator Writing System from the Development of CAD Software for LSI Design", Proceedings- IEEE International Conference on Circuits and Computers, ICCC 82, IEEE Press, 1982.

Kowalski, T.J. and Thomas, D.E., "The VLSI Design Automation Assistant: Prototype System", Proceedings of the 20th Design Automation Conference, 1983.

Kung, H.T., Sproull, R., and Steele, G., editors, VLSI Systems and Computations, Computer Science Press, 1981.

Kung, H.T., and Yu, S.Q., "Integrating High Performance Special Purpose Devices into a System", Proceedings SPIE, Volume 341, Real Time Signal Processing V, 1982.

Kung, S.Y., "From Transversal Filter to VLSI Wavefront Array", in Anceau, 1983, pp. 247-61.

LaPaugh, A.S., Algorithms for IC Layout: An Analytic Approach, TR-248, MIT Laboratory for Computer Science, 1980.

Lauther, U., Cell Based VLSI Design System, in [Rabbat, 1983, pp. 480-494].

Lee, B., Ritzman, D., and Snapp, W., "Silicon Compiler Teams with VLSI Workstation to Customize CMOS ICs", Electronic Design, Volume 32, Number 23, 15 November 1984.

Lee, C.H., and Lin-Hendel, C.G., "Current Status and Future Projection of CMOS Technology", Proceedings IEEE COMPCON, Fall 1982.

Leighton, F.T., and Leiserson, C.E., "Wafer-scale Integration of Systolic Arrays", Proceeding 23rd IEEE Symposium of Foundations of Computer Science.

Leinwand, S., and Lamdan, T., "Design Verification Based on Functional Abstraction", Proceedings 16th Design Automation Conference, 1979.

Leiserson, C.E., Rose, F.M., and Saxe, J.D., "Optimizing Synchronous Circuits by Retiming", in Bryant, 1983, pp 87-116.

Liesenberg, H.K.E., and Kinniment, D.J., "Autolayout System for a Hierarchical I.C. Design Environment", Integrated VLSI, Volume 1, Numbers 2-3, October 1983.

Lopez, A.D., and Law, H.F., "A Dense Gate Matrix Layout Style for MOS LSI", Digest of Technical Papers, ISSCC 1980.

Louie, G., Ho, T., and Cheng, E., "The Microvax I Data-Path Chip", VLSI Design, Volume 4, Number 8, December 1983.

Lowe, L., "VLSI Design Shrinks to Mere Man-Weeks", Electronics, 1982.

Lukuhay, J. and Kubitz, W.J., "A Layout Synthesis System for nMOS Gate-Cells", Proceedings of the 19th Design Automation Conference, 1982.

Lyon, R.F., A Bit-Serial VLSI Architectural Methodology for Signal Processing, in Gray, 1981, pp. 131-140.

Lyon, R.F., MSSP: A Bit-Serial Multiprocessor for Signal Processing, in Capello, 1984, pp. 64-75.

Naval Postgraduate School Technical Report NPS52-81-009, The Structural Analysis of Programming Languages, by B.J. MacLennan, 9 June 1981.

Naval Postgraduate School Technical Report NPS52-81-013, Programming with Relational Calculus, by B.J. MacLennan, 3 September 1981.

Naval Postgraduate School Technical Report NPS52-83-010, A Computer Science Version of Gödel's Theorem, by B.J. MacLennan, August 1983.

Naval Postgraduate School Technical Report NPS52-83-012, Relational Programming, by B.J. MacLennan, September 1983.

Naval Postgraduate School Technical Report NPS52-83-013, A Commentary of Mill's Logic, Book I- Of Names and Propositions, by B.J. MacLennan, October 1983.

Markov, L.A., Fox, J.R., Blank, J.H., "Optimization Techniques for Two-Dimensional Placement", Proceedings ACM IEEE 21st Design Automation Conference, June 1984.

Matheson, T.G, Buric, M.R., and Christensen, C., "Embedding Electrical and Geometric Constraints in Hierarchical Circuit-Layout Generators", International Conference on Computer-Aided Design, 1983.

Mathews, R., Newkirk, J., and Eichenberger, P., "A Target Language for Silicon Compilers", Digest of Papers Spring COMPCON 82. High Technology in the Information Industry, IEEE Computer Society, 1982.

Mazor, S., "Data Path Design Shows Worth of Silicon Compiler to VLSI System Makers", Electronic Design, Volume 32, Number 20, 4 October 1984.

Mead, C.A. and Lewicki, G., "Silicon Compilers and Foundries Will Usher in User-Designed VLSI", Electronics, Volume 55, Number 16, 11 August 1982.

Mead, C.A., "Structural and Behavioural Composition of VLSI", in Anceau, 1983, pp. 3-8.

Mennear-Dubas, S., "Silicon Compilers Adds System Based on Silicon Compilation", Computer Systems News, No. 186, 5 November 1984.

Mostow, J., "A Desision-Based Framework for Understanding Hardware Compilers", Journal of Systems and Software, Volume 4, Number 1, April

1984; also in Workshop Report: VLSI and Software Engineering Workshop, 1984, pp. 117-125.

Moulton, A., "Laying the Power and Ground Wires on a VLSI Chip", ACM IEEE 20th Design Automation Conference, 1983.

Murray, A.F., Denyer, P.B., and Renshaw, D., "Self-Testing in Bit Serial VLSI Parts: High coverage at Low Cost", Proceedings IEEE International Test Conference, October 1983.

Nash, J.H., and Smith, S.G., "A Front End Graphics Interface to the FIRST Silicon Compiler", IEE European Conference on Electronic Design Automation, March 1984.

NewKirk, J. A. and Matthews, R., The VLSI Designer's Library, Addison-Wesley, 1983.

Offen, R.J., VLSI Image Processing, McGraw Hill, 1985.

Organick, E.I., Lindstrom, G., Smith, D.K., Subrahmanyan, P.A., and Carter, T., Transformation of ADA Programs into Silicon, Semiannual Technical Report UTEC 82-020, University of Utah, 1982.

Ostreicher, D., "Where is Computer-Aided Design Going?", Computer, Volume 16. Number 5, May 1983.

Panasuk, C., "Silicon Compilers Make Sweeping Changes in the VLSI Design Worlds", Electronic Design, Volume 32, Number 19, 20 September 1984.

Parker, A.C. et. al., "The CMU Design Automation System: An Example of Automated Data-Path Design", Proceedings of the 16th Design Automation Conference, 1979.

Pearl, J., Heuristics: Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley, 1984.

Percival, R., and Fitchett, M., "Designing a Laser-Personalized Gate Array", VLSI Design, Volume 5, Number 2, February 1984.

Peskin, J.G., "Towards a Silicon Compiler", Proceedings of the 1982 Custom Integrated Circuits Conference, IEEE Press, 1982.

Pope, S. and Broderson, R., "Macrocell Design for Concurrent Signal Processing", in Bryant, 1983, pp. 395-412.

Pope, S., Rabaey, J. and Broderson, R., "Automated Design of Signal Processors Using Macrocells", in Capello, 1984, pp. 239-251.

Pope, S., Automated Generation of Signal Processing Circuits, Ph. D. Dissertation, Memo UCB/ERL M85/11, U.C. Berkeley, 1985.

Posa, J.G., "Superchips Face Design Challenge", High Technology, Volume 3, Number 1, 1983.

Poundstone, W., The Recursive Universe, Contemporary Books, Inc, 1985.

Powell, P.A.D., and Elmasry, M.I., "The Icewater Silicon Compiler", 1983 IEEE International Symposium on Circuits and Systems, Volume 2, IEEE Press, 1983.

Rabaey. J., Lager: An Automated Layout Generating System for Digital Signal Processing, User Manual V1.3, Memo UCB/ERL M85/5, U.C. Berkeley, 1985.

Rabaey, J., Pope, S.P., Broderson, R.W., "An Integrated Automated Layout Generation System for DSP Circuits", IEEE Transactions on Computer-Aided Design, Volume CAD-4, Number 3, IEEE Press1985.

Rabbat, G., editor, Hardware and Software Concepts in VLSI, Van Nostrand Reinhold Company, 1983.

Randell, N., and Treleaven, P.C. (editors), VLSI Architecture: 1982 Advanced Course on VLSI, Prentice-Hall, 1983.

Rattner, J.R., "Functional Extensibility: Making the World Safe for VLSI", in Kung, 1981, pp. 50-51.

Ravi, N., Bruss, A. and Reif, J., "Linear Time Algorithms for Optimal CMOS Layout", in Bertolazzi, 1985, 327-338.

Reekie, H.M., Mavor, J., Petrie, N., and Denyer, P.B., "An Automated Design Procedure for Frequency Selective Wave Filters", 1983 IEEE International Symposium on Circuits and Systems, Volume 1, IEEE Press, 1983.

Reekie, H.M., Petrie, N., Mavor, J., Denver, P.B., and Lau C.H., "Design and Implementation of Digital Wave Filters Using Universal Adaptor Structures", IEE Proceedings 1984, Part F, Volume 131, Number 6, 1984.

Reiss, S.P. and Savage, J.E., "SLAP- A Silicon Layout Program", Proceedings IEEE International Conference on Circuits and Computers, 1982.

Reutz, P.A., Pope, S.P. Solberg, B., and Broderson R.W., "Computer Generation of Digital Filter Banks", Digest of Technical Papers, IEEE International Solid State Circuits Conference (ISSCC-84), February 1984.

Rivest, R.L., "The PI (Placement and Interconnect) System", Proceedings 20th Design Automation Conference, 1982.

Rosenberg, A.L., "References to the Literature on VLSI Algorithmics and Related Mathematical and Practical Issues", Sigact News, Volume 16, Number 3, Fall 1984.

Rosenberg, J.B., "Chip Assembly Techniques for Custom IC Design in a Symbolic Virtual Grid Environment", Proceeding MIT Conference on Advanced Research in VLSI, January 1984.

Rosenberg, J.B., and Weste, N.H., ABCD- A Better Circuit Description, MCNC Technical Report 4983-01, Microelectronic Center of North Carolina, 1983.

Rupp, C.A., "Components of a Silicon Compiler System", in Gray, 1981, pp. 227-236.

Saucier, G., and Serrero, G., "Intelligent Assistance for Top Down Design of VLSI Circuits", in Workshop Report: VLSI and Software Engineering Workshop, 1984, pp. 107-111.

Savage, J.E., "Three VLSI Compilation Techniques: PLAs, Weinberger Arrays, and SLAP, (A New Silicon Layout Program)", ALgorithmically-Specialized Computers, 1983.

Schindler, M., "Transforming a Simulated IC Into Silicon Demands Tools That Unravel Design Decisions", Electronic Design, Volume 32, Number 25, 13 December 1984.

Schoellkopf, J.P., "LUBRICK: A Silicon Assembler and its Application to Data-Path Design for FISC", in Anceau, 1983, pp. 435-45.

Schuck, J., Glesner, M. and Joepen, H., ALGIC: A Flexible Silicon Compiler System for Digital Signal Processing, VLSI Signal Processing, IEEE Press, 1984.

Shenton, G., "The Importance of Design Methodology in CAD Tool and Vendor Selection", SOUTHCON/84 Electronics Show and Convention, January 1984.

Shrobe, H.W., "The Data Path Generator", MIT Conference on Advanced Research in VLSI, Spring 1982.

"Silicon Compilers I: Drawing a Blank", VLSI Design, Volume 5, Number 9, September 1984.

"Silicon Compilers has VLSI Design System", Electronic News, Volume 30, No. 1518, 8 October 1984.

"Silicon Compilation Said to Speed VLSI Circuit Design", Computerworld, Volume 18, Number 41, 8 October 1984.

Smith, K., "Silicon Compiler Cuts Design-to-Chip Time for Large Gate Arrays", Electronics, Volume 56, Number 23, 17 November 1983.

Smith, D., "Turnkey VLSI IC-Design System Uses Silicon Compilation", EDN, Volume 29, Number 20, 4 October 1984.

Smith, C.U., and Dallen, J.A., "A Comparison of Design Strategies for Software and for VLSI", in Workshop Report: VLSI and Software Engineering Workshop, 1984, pp. 160-165; and Proceedings COMCON (Spring) '83, 1983.

Southard, J.R., "Silicon Compiler Demands No Hardware Expertise to Fashion Custom Chips", Electronic Design, Volume 32, Number 23, 15 November 1984.

Srini, V.P., "Test Generation for MacPitts Designs", Proceedings IEEE International Conference on Computer Design, (ICCD-83), 1983.

Stefik, M. et. al., "The Partitioning of Concerns in Digital System Design", MIT Conference on Advanced Research in VLSI, 1982.

Subrahmanyan, P.A., "Synthesizing VLSI Circuits from Behavioral Specifications: A Very High Level Silicon Compiler and Its Theoretical Basis", in Anceau, 1983, pp. 195-210.

Suzim, A.A., "Data Processing Section for Microprocessor-Like Integrated Circuits", IEEE Journal of Solid State Circuits Conference, 1981.

Swartzlander, E.E., VLSI Signal Processing Systems, Kluwer Academic Publishers, 1985.

"The Evolution of Chip Customization", High Technology, Volume 2, Number 1, January 1983.

Touretzky, D. S., LISP: a Gentle Introduction to Symbolic Computation, Harper & Row, 1984.

Travassos, R.H., "Hardware Design Automation", Proceedings of the 1983 American Control Conference, September 1983.

Trickey, H.W., "Good Layouts for Pattern Recognizers", IEEE Transactions on Computing, Volume C-31, Number 6, June 1982.

Trickey, H.W. and Ullman, J.D., "Regular Expression Compiler", Digest of Papers- COMPCON Spring 82: High Technology in the Information Industry, IEEE Computer Society, 1982.

Turner, L.E., Denyer, P.B., and Renshw, D., "A Bit Serial LDI Recursive Digital Filter", Proceedings of the IEEE Internation Conference on Acoustics, Speech and Signal Processing (ICASSP 84), March 1984.

Ullman, J. D., Computational Aspects of VLSI, Computer Science Press, 1984.

VanCleemput, W.S., "Computer Hardware Description Languages and Their Applications", Proceedings 16th Design Automation Conference, 1979.

Vandewalle, J., et. al., "A Unified Box of VLSI Building Tools for Digital Signal Processors", Proceedings ICCD 1984, 1984.

Wallich, P., "On the Horizon: Fast Chips Quickly", IEEE Spectrum, Volume 21, Number 3, March 1984.

Weinberger, A., "Large Scale Integration of MOS Complex Logic: A Layout Method", IEEE Journal of Solid State Circuits, Volume SC-2, 1967.

Werner, J., "The Silicon Compiler: Panacea, Wishful Thinking, or Old Hat?", VLSI Design, Volume 3, 1982.

Werner, J., "Progress Toward the 'Ideal' Silicon Compiler Part 1: the Front End", VLSI Design, Volume 4, Number 5, 1983.

Werner, J., "Progress Toward the 'Ideal' Silicon Compiler Part 2: the Layout Problem", VLSI Design, Volume 4, Number 6, 1983.

Weste, N. and Eshragian, K., Principles of CMOS VLSI Design: a Systems Perspective, Addison-Wesley, 1985.

Wieclawski, A., and Perkowski, M., "Optimization of Negative Gate Networks Realized in Weinberger-Like Layout in a Boolean Level Silicon Compiler", Proceedings ACM IEEE 21st Design Automation Conference, June 1984.

Williams, J.D., STICKS: A New Approach to LSI Design, Master's Thesis, Massachusetts Institute of Technology, 1977.

Williams, J.D., "STICKS- A Graphical Compiler for High Level Design", AFIPS Conference Proceedings, Volume 47, 1978.

Williams, T.W., and Parker, K.P., "Design for Testability: A Survey", IEEE Transactions on Computers, January 1982.

Winston, P. E., Artificial Intelligence, 2d ed., Addison-Wesley, 1984.

Wittenberg, R.C., "Daisy, Silicon Compilers Sign Cooperation Pact", Electronic Engineering Times, No. 306, 19 November 1984.

Wolf, W., Newkirk, J., Mathews, R., and Dutton, R., "Dumbo: A Schematic-to-Layout Compiler", in Bryant, 1983, pp. 379-391.

Workshop Report. VLSI and Software Engineering Workshop, IEEE Computer Society Press, 1984.

Young, J., "Why Silicon Compilers had to Change its Strategy", Electronics, 1985.

Zarrella, J., "How Silicon Compilation will Affect Engineers", Microcomputer Applications, COMPCON Spring '85 Computer Conference, IEEE Computer Society, 1985.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Department Chairman, Code 62 Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	1
4.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	1
5.	Dr. D. E. Kirk, Code 62KI Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	8
6.	Dr. H. H. Loomis, Code 62LM Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	3
7.	Prof. B. J. MacLennan, Code 52MI Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	1

8. Prof. R. McGhee, Code 52Mz — 1
Department of Computer Science
Naval Postgraduate School
Monterey, California 93943-5000

9. Prof. D. Bukofzer, Code 62BH — 1
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, California 93943-5000

7. Mr. P. Blankenship — 1
Massachusetts Institute of Technology
Lincoln Laboratory
P.O. Box 73
Lexington, Massachusetts 02173-0073

8. Mr. J. O'Leary — 1
Massachusetts Institute of Technology
Lincoln Laboratory
P.O. Box 73
Lexington, Massachusetts 02173-0073

9. Mr. A. Casavant — 1
Massachusetts Institute of Technology
Lincoln Laboratory
P.O. Box 73
Lexington, Massachusetts 02173-0073

10. Dr. C. Sequin — 1
Associate Chairman
Computer Science Division
University of California
589 Evans Hall
Berkeley, California 94720

11. LTCOL H. W. Carter, USAF — 1
Air Force Institute of Technology
Department of Electrical Engineering
AFIT/ENG Building 640 Area B
Wright-Patterson Air Force Base, Ohio 45433

12. Dr. D. Gajski 1
Department of Computer Science
210 Digital Computer Laboratory
1304 Springfield Avenue
Urbana, Illinois 61801

13. MAJ E. Weist, USMC 1
Naval Pacific Missile Test Center
AATN: Marine Aviation Detachment XO
Point Mugu, California 93042

14. LCDR J. Harmon, USN 1
SMC #2231
Naval Postgraduate School
Monterey, California 93943

15. LT A. Mullarky, USN 1
SMC #1424
Naval Postgraduate School
Monterey, California 93943

16. Dr. T. Bestul 1
Naval Research Laboratories
Code 7590
Washington, D.C. 20375

17. Dr. D. O'Brien 1
Lawrence Livermore National Laboratory
P.O. Box 5504, L-156
Livermore, California 94550

18. Mr. A. DeGroot 1
Lawrence Livermore National Laboratory
P.O. Box 808,
Livermore, California 94550

19. Mr. E. Carapezza, Code 52 — 1
Department of Computer Science
Naval Postgraduate School
Monterey, California 93943-5000

20. Dr. F.A. Malagon-Díaz — 1
2998 Plaza Blanca
Santa Fe, New Mexico 87505-5340

21. CAPT, E. Malagón, USMC — 1
SMC #2480
Naval Postgraduate School
Monterey, California 93943

22. LCDR M. A. Malagón-Fajar, USN — 1
1220 7th Street, #2
Monterey, California 93940

23. Prof. H. Titus, Code 62Ts — 1
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, California 93943-5000

24. Prof. S. Michaels, Code 62Ms — 1
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, California 93943-5000

25. Prof. L. Abbott, Code 62At — 1
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, California 93943-5000

218807

Thesis
M27765 Malagon-Fajar
c.1 Silicon compilation
using a LISP-based
layout language.